Analysis and Simulation of Emergent Architectures for Internet of Things

Damián Roca Marí

Computer Architecture Department (DAC)

Universitat Politècnica de Catalunya

A dissertation submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy

December 2017

Dedicated to my family for believing in me since the first day

Acknowledgements

First of all, I want to express my most sincere gratitude to my advisors Mario Nemirovsky and Mateo Valero. Mario's positive and supporter character helped me to overcome obstacles and become smarter. He always makes the right questions, showing an exceptional criticism that formed me as a researcher. The freedom he gave me allowed me to develop my skills and to expand my vision about everything. He is not only an advisor but a close friend. Mateo took me into the BSC and made me feel respected and supported, providing the perfect environment to grow and pursue my PhD. His advices are always great, revealing that he is not only an excellent researcher but also an excellent person. I thank them for their confidence, support, and trust in me since my first day. I grew up a lot during this journey with you.

A special thanks to Rodolfo Milito, who I consider an advisor to my thesis and a great friend. Thanks for sharing this journey with us and showing that structure and rational analysis are not lost. This thesis would not be the same without you. Many thanks to Andres Gomez, my mentor and my friend, for sharing many hours working and coding. He was the people who showed me what being an engineer means, and highlighted the path that took to me to this incredible journey that is only starting ahead of me.

I would like to thank Prof. Reza Nejabati, Prof. Fulvio Risso, and Prof. Ramon Canal for serving in my thesis committee, providing very useful comments and suggestions that improved this thesis.

I am indebted to my colleagues and friends from Barcelona Supercomputing Center (BSC) sharing deadlines and long hours working,

without you it would not have been the same. Many thanks to Josue Quiroga, Gabriel Fernandez, Francesco Ciaccia, Daniel Nemirovsky, Oscar Palomar, Mladen Slijepcevic, Javier Arias, Timothy Hayes, Milan Stanic, Omer Subasi, Ivan Ratkovic, Ferad Zyulkyarov, Behzad Salami, Adria Armejach, Sasa Tomic, Nehir Sonmez, Adrian Cristal, Osman Unsal, Emma Torrella, Guadalupe Moreno, Xavi Salazar, Oriol Arcas, Srdjan Stipic, Vesna Smiljkovic, Azam Seyedi, Nikola Markovic, Gulay Yalcin, Vladimir Subotic, Oyku Melikoglu, Kadir Tugberk Arkose, Burcu Mutlu, Vasilis Karakostas, and Pavlos Maniotis. Also thanks to the administrative teams at BSC and UPC for their help.

I would like to thank the people who made possible my internship in Los Gatos, an enriching experience that changed me. Thanks to Laura Nemirovsky for accepting me into her home, and many thanks to all my friends there who made me feel like at home.

I am grateful to the Fundación La Caixa for their PhD scholarship. Without it, my PhD would not had been the same. Sharing many events, talks, and dinners I have met outstanding people and made many friends, becoming part of your family. A special mention to people at the scholarship team for their support and assistance during this journey, thanks.

Finally, I would like to thank my friends and family for being here during this journey. My deepest gratitude to Danielle for being her, for her love, and for our amazing time together that showed me what life truly means. One stage is closing and a new exciting one is starting.

Abstract

The Internet of Things (IoT) promises a plethora of new services and applications supported by a wide range of devices that includes sensors and actuators. To reach its potential IoT must break down the silos that limit applications' interoperability and hinder their manageability. These silos' result from existing deployment techniques where each vendor set up its own infrastructure, duplicating the hardware and increasing the costs. Fog Computing can serve as the underlying platform to support IoT applications thus avoiding the silos'.

Each application becomes a system formed by IoT devices (i.e. sensors, actuators), an edge infrastructure (i.e. Fog Computing) and the Cloud. In order to improve several aspects of human lives, different systems can interact to correlate data obtaining functionalities not achievable by any of the systems in isolation. Then, we can analyze the IoT as a whole system rather than a conjunction of isolated systems. Doing so leads to the building of Ultra-Large Scale Systems (ULSS), an extension of the concept of Systems of Systems (SoS), in several verticals including Autonomous Vehicles, Smart Cities, and Smart Grids. The scope of ULSS is large in the number of things and complex in the variety of applications, volume of data, and diversity of communication patterns.

To handle this scale and complexity in this thesis we propose Hierarchical Emergent Behaviors (HEB), a paradigm that builds on the concepts of emergent behavior and hierarchical organization. Rather than explicitly program all possible situations in the vast space of ULSS scenarios, HEB relies on emergent behaviors induced by local

rules that define the interactions of the "things" between themselves and also with their environment.

We discuss the modifications to classical IoT architectures required by HEB, as well as the new challenges. Once these challenges such as scalability and manageability are addressed, we can illustrate HEB's usefulness dealing with an IoT-based ULSS through a case study based on Autonomous Vehicles (AVs). To this end we design and analyze well-though simulations that demonstrate its tremendous potential since small modifications to the basic set of rules induce different and interesting behaviors. Then we design a set of primitives to perform basic maneuver such as exiting a platoon formation and maneuvering in anticipation of obstacles beyond the range of on-board sensors. These simulations also evaluate the impact of a HEB deployment assisted by Fog nodes to enlarge the informational scope of vehicles.

To conclude we develop a design methodology to build, evaluate, and run HEB-based solutions for AVs. We provide architectural foundations for the second level and its implications in major areas such as communications. These foundations are then validated through simulations that incorporate new rules, obtaining valuable experimental observations.

The proposed architecture has a tremendous potential to solve the scalability issue found in ULSS, enabling IoT deployments to reach its true potential.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we first present the fundamental characteristics of the Internet of Things. We then discuss the motivation behind our work and state the problems that we tackle in this thesis. Then we present our approach for each of these problems, and, finally, we provide an overview of our contributions that enhance IoT to reach its true potential creating a collaborative environment with new marketplaces.

1.1 Internet of Things

Internet of Things (IoT) includes a pervasive presence of sensors, actuators, and other devices that are deployed across large areas and connected via protocols (e.g. Bluetooth, WiFi, LoRA, 5G) that cooperate to meet common objectives. The dominant characteristic of IoT is the physical interaction of the "things" with their environments, which enables novel applications and sets new architectural demands. Through the deployment of trillions of "things", IoT is significantly transforming and improving city services, transportation, agriculture, health-care, energy production and distribution, and water conservation, among many other vital aspects of human life.

Many of these applications are widely distributed, some have stringent real time requirements, and in all cases it is necessary to maintain trustworthy communication and adaptability to dynamic environments. "Things" require an infrastructure on top of them capable of satisfying the aforementioned requirements,

enabling a plethora of IoT-based applications. This platform allows to exploit the visibility that "things" provide because, in most cases, "things" have reduced capabilities (i.e. limited computing capabilities and storage). Hence, the platform operating on top of the "things" becomes a critical part of each IoT system.

Despite the heterogeneity of IoT applications, architects would desire a single infrastructure capable of satisfactory respond to any application requirements set. Cloud Computing can support a subset of the IoT applications, but characteristics such as a centralized platform, distance to the "things", and connectivity precludes real-time and/or critical applications. Different platforms appeared in the last years to provide a solution for these applications while simultaneously exploit the advantages of the Cloud. Fog Computing [15] constitutes a remarkable architecture that operate at the edge of the network to satisfy real time applications among others.

Fog Computing can be used as a base to provide an infrastructure that enables that interoperability. It has the potential to become the "de facto" IoT platform since it is capable of satisfying the applications requirements. It is a highly distributed platform with nodes located from near the "things" till the edge of the network. These nodes offer computation, storage, and networking capabilities to the applications operating beneath its infrastructure. Fog processes the data close to where its generated, reducing the network utilization and improving the aggregation from the bottom of the infrastructure. Low-latency, widespread geographic distribution, heterogeneity, and mobility are part of its main advantages.

Current deployment techniques and the lack of a generic platform to execute and support IoT applications end up creating silos, where each application deploys its own hardware. Fog eliminates the silos since each of these systems is formed by a set of sensors and actuators together with instances of a generic Fog-based platform (and optional support from the Cloud), avoiding the hardware duplicity. For example, one system can provide smart mobility where public transportation adapts to the user's needs in real time (i.e. assign vehicles from low utilization routes to others with high demand). In parallel, another system can focus on providing efficient routes to vehicles, avoiding congestions and thus reducing the

time to destination and saving fuel. Both applications can be executed under Fog's infrastructure, eliminating silos.

Managing a current IoT sub-system with a low number of devices already poses many threats due to the amount of resources required to monitor devices and obtain useful information. The problem aggravates when considering all IoT sub-systems within an area such as transportation as a single entity. The resultant system is in fact an IoT-based Ultra Large Scale System (ULSS), that is an assemblage of different components where each is both operationally and managerially independent. Scalability and manageability are major concerns when observing all those IoT subsystems as an ULSS, but also complexity and orchestration.

For instance, if we focus our attention on one of these applications within the field of smart transportation, Autonomous Vehicles (AVs), we can analyze the different independent systems that conform this ULSS. Vehicles in all its variants conform different systems where each vendor or application defines a subsystem. Amazon could use a drone system to deliver its packages while a government could use another drone system to monitor the traffic. Both of these systems are independent but they need to understand each other. However, the potential functionalities that can emerge when both work in the same geographical area are still not envisioned. There is plenty of room to exploit their convergences and enable new applications that each system in isolation could not provide. This is just an illustrative example, but the amount of systems interacting to enable AVs is much larger, including control systems (i.e. traffic lights), external regulations (i.e. traffic rules), and human-driven vehicles among others.

There are compelling reasons to decentralize ULSS. They include manageability, scalability, complexity that grows with the scale of the system; and the ability to contain failures. All together stresses the scalability of the system due to the stringent number of IoT devices envisioned. Even though each device can be really simply (i.e. temperature sensor), when millions of them generate data the resulting system turns to be quite complex.

Concerning the open literature, there is no consensus on a technique that tackles the scalability and complexity of the ULSS IoT. Current IoT systems rely

upon extensive coding to achieve an explicitly-programmed behavior. This technique does not scale since each new functionality of the ULSS requires extensive programming, limiting the space for new functionalities to emerge.

1.2 Motivation

Despite the fact that IoT's potential has been stated since its origins, IoT applications and their functionalities have not reached those expectations yet. Several reasons are responsible of this delay including but not limited to current "silo" based deployments, lack of a clear monetization channel, Capex and Opex, security mechanisms, and data confidentiality.

Fog presents an elegant solution to some of these problematics, but requires that stakeholders own vertical deployments from the "things" to the Cloud. In consequence, there is a great entry barrier for new IoT applications. However, Fog nodes need to become a generic platform to allow IoT systems to reach their potential and thus helps to bring a real democratization to the IoT domain. The first step in this direction would the elimination of "silos", followed by a series of software APIs to facilitate the usage of the hardware infrastructure (i.e. in the form of virtual instances).

Once IoT systems exploit, a solution that tackles the complexity and the scalability of the resultant ULSS is required. Currently there are no consensus on how this objective should be achieved, and existing systems rely on explicitly-programmed behaviors. This technique, although partially satisfactory does not scale and presents huge development costs. The behavior of the "things" is defined from the conception of the system, and this poses a thread over the scalability of the system. In addition, they need to take into account an overwhelming number of scenarios that have to be coded or elsewhere applications are not capable of adapting to their environments. Aggravating the problem, the open literature focus on solutions centered in single elements of the system, such as a vehicle, rather than considering the perspective of the system that millions of vehicles conform.

For IoT to reach its potential a new multidisciplinary approach is required to tackle the aforementioned problems. Therefore, only a flexible, scalable, and

adaptive architecture could solve those issues while creating new marketplaces and business models that contribute to the expansion of IoT deployments. This solution should focus on how to enable the interoperability between different IoT systems to induce new behaviors rather than having to program explicitly.

In these emergent behaviors or functionalities is where the true potential of ULSS lies, avoiding todays limitation due to programing and managing costs of these systems. Constituent systems can be added or removed, and the only specification required is the rules of engagement between elements (i.e. car) from each system. After their interactions, new functionalities that are not explicitly programmed emerge. These behaviors exploit the contextual information and the locality of "things" to enhance the scalability of the system while reducing the managerial complexity. Then, "things" can take more decisions based on information they have rather than relying completely on their supporting platform.

1.3 Problem statement

Previous work has posed the attention over different problems that attain to the IoT ecosystem. There are two main areas we need to tackle in this thesis: (i) Fog Computing as a generic platform for IoT applications and (ii) management and orchestration of IoT ULSS applications to tackle their complexity. The first area comprises two distinct objectives since an initial assessment of the hardware required at the Fog nodes has to be performed due to the heterogeneity of the IoT applications. Once this is completed, we can focus on how to enhance Fog to become the desired generic IoT platform.

Analysis and simulation of processor architectures at the Fog node level. Fog nodes are a set of heterogeneous nodes that are interconnected forming a hierarchy. Taking a more detailed look to the first aggregation level, those nodes directly connected to the "things", they need to execute a wide range of applications with different requirements over their hardware platform. Some applications may require more memory access and low computational power while others may focus on real time execution times to ensure their safety and criticality.

The first goal of this thesis is to develop a new hardware and software simulation tool that allows researchers to perform fast but still accurate design space

exploration analysis. Then, researchers can quickly explore what type of hardware better fits their application requirements before proceeding to a more detailed hardware/software simulation if required. Instead of focusing on a single application, this thesis looks for a generic simulation platform that can be used for the forthcoming IoT applications

Fog Computing enhancements to become a generic platform. Fog characteristics place this platform as a suitable candidate to enable IoT applications. However, due to the nature of its current deployments, each application needs to develop complex software solutions to integrate the nodes or to afford vertical deployments (from "things" to the Cloud) that only large companies can undertake.

The second goal of this thesis is to provide enhancements to Fog's platform so these complex software solutions are no longer required. Hence, Fog would become a true generic platform where each application only requires an instance to start running. This fact would also enable a democratization of the IoT platform, reducing the entry barriers for new IoT-based services running on the edge of the network.

Tackle the complexity of IoT ULSS. Even though a generic infrastructure can support IoT services, applications are still heavily complex due to the number of "things" involved and the complex scenarios where they are deployed. Managing and orchestrating these applications with traditional solutions is not very effective since they rely on explicitly programmed software solutions that focus on the perspective of every device. In this thesis we focus on AVs as it constitutes a prime example of ULSS where the open literature focuses on single vehicles rather the system perspective.

The third goal of this thesis is to analyze the requirements of an ULSS AV system, and to provide a solution to IoT ULSS that is scalable and feasible to program. This architecture should empower the "things" to reduce the pressure placed upon the infrastructure, exploit the contextual information to enrich IoT-based services, and adapt to complex environments under continuous change.

1.4 Thesis approach

In this thesis we develop a multidisciplinary approach to develop a simple but powerful architecture that englobes the infrastructure and the application to exploit the best from both worlds. To design this transversal solution, we first focus on Fog Computing as the "de facto" IoT platform analyzing the hardware requirements at the first aggregation levels and possible enhancements to make of Fog a generic IoT platform capable of executing any type of application without the need of new hardware deployments. Second, we focus on the application side to orchestrate ULSS improving their scalability inspired by concepts from other areas of knowledge such as ant colonies.

To this end, we introduce the concept of Hierarchical Emergent Behaviors, inspired in flocks of birds and schools of fish, to design an AV system. We show that the combination of emergent behaviors, those not explicitly programmed, and hierarchical decomposition tackles the complexity of the ULSS by leveraging the decision-making to the "things" themselves, those devices with a natural access to their contextual information. Hence, we argue that this technique is the way to design and implement AV systems based on extensive simulations and case studies.

Despite the fact that the assessment of the Fog nodes' hardware – to which end we designed a simulation tool – may seem disjoint, IoT is a vast area with many different verticals. We evaluated the hardware platform on our first approach to the IoT domain to enable new IoT applications exploring the applications' infrastructure requirements. HEB solves a wider problem in dealing with the scalability, complexity, and manageability of an AV ULSS rather than focusing on the specifications of a node within the ULSS.

1.5 Thesis contributions

This thesis makes the following contributions:

1. **A simulation methodology based on queue models and statistical information (named iQ)**, targeting design space exploration analysis on

the early stages of the processor design process. This hardware defines the Fog nodes, that form an underlying platform to support and execute IoT workloads. Given the wide range of IoT applications and due to the lack of IoT-based benchmarks, we evaluate our simulation tool with the spec cpu2006 to present the methodology that, once IoT applications are available, can be applied to obtain fast results. We find that by emulating key processor components using queue structures we can obtain a very accurate result with a significant speedup. The code is available at this repository [3].

2. **Enhancements to Fog's architecture to enable the generic IoT platform.** We propose three enhancements targeting three critical areas: (i) a new orchestration policy, (ii) the creation and usage of constellation of Fog nodes, and (iii) the definition of Fog Function Virtualization (FFVs). The new orchestrator gives flexibility to the infrastructure to truly exploit the Fog layers. To mitigate the current lack of resources at the Fog node level, we introduce constellations of Fog nodes. They are virtual groupings of nodes to aggregate their capabilities. Finally, FFVs tackle the problem of the complex software solutions required nowadays faced when deploying new applications.

3. **A design methodology based on emergent behaviors and hierarchical decomposition for ULSS, named Hierarchical Emergent Behaviors (HEB).** HEB tackles the scale and complexity of ULSS proposing a paradigm change from explicitly-programmed applications to the application of simple but powerful local rules at the "things" level. The key idea is to induce self-organizing behaviors akin to the swarm formation, thus bypassing arduous, centralized, and potentially brittle control mechanisms. A well-designed HEB promises to be more flexible and adaptable to unanticipated conditions than a traditionally hard coded system. To this end, we analyzed its characteristics and future potential through carefully crafted simulations focused on Autonomous Vehicles (AVs) case studies. The code is available at this repository [2].

Next we highlight the most important concepts of each contribution.

1.5.1 iQ

Computer architects use different simulation methodologies to design, evaluate and optimize computing systems. The selection process becomes non-trivial due to the number of available tools. An extended technique relies on full cycle simulations with tools such as Gem5 [13] or MARSS [52]. They provide a detailed accuracy but at the cost of a tremendous development complexity. Another solution uses trace-driven simulators to reduce the execution time, but its complexity remains the same [38, 55, 77]. The only technique that does not bring the complexity consists of using analytical models. Its development effort is small, but sacrificing part of the accuracy. There is no generic tool or methodology to perform design space exploration analysis on a fast but still accurate method to identify critical areas and bottlenecks that drive the processor architecture design process.

In this thesis, we design and develop a methodology to perform design space exploration analysis on fast manner while maintaining the accuracy. Our tool, named iQ, builds on queue models and statistical information to construct models that emulate the behavior of real hardware. Then, architects can effectively a two-stage process to first identify bottlenecks and later focus on more finer granularity simulation. During this first stage, a hardware/software co-design process drives the analysis thanks to the abstraction brought by the queue models. Since there is not a "de facto" IoT benchmark, we evaluated iQ against the well-known SPEC CPU2006 suite to evaluate our methodology and its potential.

We find that iQ's modular nature and easy reconfigurability of the component parameters make results in a highly flexible and powerful processor simulator. We observe that this technique is capable of accurately represent modern single core processors with a remarkable accuracy. In addition, it provides an almost ISA independent capable of emulating not only processors but also FPGAs and GPUs.

We have built an Ivy Bridge and a Core 2 Duo processor model and have validated them against real hardware running SPEC CPU2006 Int achieving average error rates of 9.55% and 8.93%, proving that there are efficient alternatives to full system simulators when it comes to design space exploration studies.

Finally, we perform a design space exploration taking the Ivy Bridge as base to determine what processor components can be modified to increase its instruction per cycle for the chosen benchmark suite. An important remark arises from the insight architects can gain by using iQ on where the real bottlenecks of a modern and complex processor are.

1.5.2 Fog Computing Enhancements

Fog Computing [15] is a highly distributed platform that relies on nodes located at the edge of the network, closer to the end-user devices, to provide computation, storage, and network resources. Access points and routers constitute clear examples of Fog nodes, although companies such as Nebbiolo are designing specific hardware to operate as Fog nodes. Thanks to its characteristics including but not limited to low-latency, wide-spread geographic distribution, heterogeneity, and mobility Fog is positioning itself as the "de facto" IoT platform.

For Fog to achieve its potential it needs to become a generic platform capable of serving multiple applications simultaneously. However, current IoT deployments are based on "things" covering a geographical area with a set of proprietary nodes connected to the Internet. As a consequence, each application constitutes a subsystem or silo inside IoT, avoiding data correlation between different applications that could benefit the users. On top of this system isolation, complex software solutions are required to integrate all the components of each system (i.e. sensors, nodes, cloud), forcing companies to own the complete vertical stack from "things" till the Cloud.

In this thesis, we propose three enhancements to Fog: (i) a new orchestration policy to provide more flexibility to the infrastructure breaking the execution in the Cloud by default, (ii) the creation of constellations of heterogeneous Fog nodes to aggregate their capabilities, and (iii) the definition of the Fog Function Virtualization (FFV) concept.

The new orchestration policy breaks the default execution on the Cloud and allow to further exploit the Fog layers whenever the requirements allow it. The constellations create the virtual image of having more resources at the lower level

of the hierarchy without the need of deploying new hardware by aggregating resources from close nodes. FFVs allow to offer infrastructure resources to the applications without worrying about the underlying hardware. For instance, sensors can be offered as a service, or we can define functions at the Fog node levels such as combination of values for a range of sensors (i.e. gather and perform the average value for a range of sensors) in a similar way as Network Function Virtualization does it for network equipment. Then, we analyze their effectiveness and potential impact through two different use cases, a smart grid scenario and a contamination information map based on smart vehicles.

Finally, we observe that the the combination of these three techniques contributes to the democratization of the IoT services by truly enabling a generic infrastructure to run multiple applications simultaneously. A new deployment only requires the application code, since virtualization techniques hide the entire infrastructure from "things" to Cloud.

1.5.3 Hierarchical Emergent Behaviors (HEB)

Internet of Things (IoT) describes the pervasive presence of sensors, actuators, and other devices that are deployed across large areas and connected via protocols (e.g. Bluetooth and WiFi) that can cooperate to solve common objectives [11]. A fundamental characteristic of IoT is the physical interaction of the "things" with their environments enabling new forms of architectures and abilities. These applications must carefully assess certain crucial factors such as the real-time and largely distributed nature of the "things", maintaining trustworthy communication, and adapting to dynamic environments. Building applications to utilize the IoT infrastructure can lead to exciting opportunities such as building smarter cities and controlling transport congestion.

Existing IoT deployments connect directly "things" to the Cloud, aggravating the "silos" problem described in the Fog Computing case. These "silos" limit the data correlation and the interoperability between different applications. The disadvantages of this implementation strategy are redundancies in hardware and in communication as well as increased deployment costs.

Though the largely exclusive manner in which IoT systems are currently utilized may ensure certain business advantages, they are inherently limited in terms of scalability, orchestration, and management. These facts threaten the ability of IoT to operate as an Ultra Large Scale System (ULSS) [50] that exploit the benefits of the underlying applications such as autonomous vehicles and smart cities.

In this thesis we propose Hierarchical Emergent Behaviors (HEB) to tackle the ULSS scale and complexity. HEB is a paradigm built on top of the concepts of emergent behaviors and hierarchical organization. The underlying concept behind emergent behaviors is to induce self-inducing behaviors through the application of a well-thought set of local rules at the "thing" level, exploiting their contextual information. Its major advantage is the bypass of arduous and complex centralized mechanisms that prevent IoT ULSS to scale to billions of devices.

Conventional IoT systems rely on explicitly-programmed behaviors achieved through complex and costly codes. Instead, our proposal relies on lightweight local rules that describe the interactions between "things" and their environment to induce those behaviors. Hence, architects do not need to anticipate all the possible scenarios reducing the design complexity while enhancing its scalability.

Unlike emergent behaviors in nature, ULSS operates at different spatial and temporal levels. In consequence, we organize our behaviors hierarchically, where level (N+1) abstracts the behavior of level (N) while widening its spatial-temporal scope.

This merge between emergent behavior and hierarchical organization concepts induces desired behaviors without the need to envisage nor explicitly program for the vast number of potential scenarios. A well-designed HEB is flexible and adaptable to unanticipated conditions than a traditionally hard coded system.

We aim to i) call the attention upon the scalability problem in IoT, ii) suggest an approach based on two well-known organizing principles, emergent behavior and hierarchical organization, and iii) stimulate future research based on the ideas and techniques related to Hierarchical Emergent Behaviors. To this end, we provide first an overview on how emergent behavior and the multi-dimensional approach applies to IoT. We then point out the architectural modifications needed in order to generate those behaviors and discuss some challenges. We illustrate

its usefulness in dealing with an IoT ULSS through a case study based on Autonomous Vehicles.

Once the fundamentals are stablished, we take a first step to validate HEB concepts through the study of two basic self-driven car primitives: exiting a platoon formation, and maneuvering in anticipation of obstacles beyond the range of on-board sensors. In this scenario, Fog nodes provide the critical contextual information required to perform those maneuvers. Hence, Fog enlarges the vision and the scope of each single vehicle in order to optimize the vehicles reactions when facing certain situations. This technique emphasizes the role of Fog Computing as support for HEB communications in general, and facilitating contextual awareness in particular.

HEB induces useful behaviors through local rules implemented at each AV rather than explicitly programming each action a vehicle must take in every circumstance. Relying on emergent behaviors has major benefits. The first is the absence of highly complex algorithms. The second is HEB's intrinsic adaptivity to deal with unanticipated corner cases and its natural scalability. These objectives are achieved by moving the decision-making capabilities to the vehicles and thus allowing them to take actions based on well understood rules.

The next logical step requires the development of a design methodology to build, evaluate, and run HEB-based solutions for AVs. This thesis advances HEB's methodology by providing Architectural foundations of the second level and its implications, with a focus on inter-level communication & locality and hierarchical relation between the rules, including the necessity of a leader and possible mechanisms to implement its selection. In addition, the AV case study is further extended to incorporate new rules and provide valuable experimental observations.

Finally, our simulations demonstrate the robustness, flexibility, and smoothness of a HEB-based AV system.

1.6 Thesis organization

Chapter 2 presents the fundamentals on the queue-based simulation methodology, focusing on an Intel Ivy Bridge model evaluated under the SPEC CPU INT 2006

benchmarks. This Chapter follows mostly from our work published in the 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2017) [59].

Chapter 3 presents an overview of Fog Computing and focuses on the three innovations developed in the content of this thesis. This Chapter follows mostly from our work published in the 2nd International Conference on Fog and Mobile Edge Computing (FMEC 2017) [58].

Chapter 4 presents the HEB concept and how it builds on the concepts of emergent behaviors and hierarchical decomposition, and its later evaluation through simulation. This Chapter follows mostly from our work published in IEEE Micro Journal Volume 36, 2016 [57] and in the Springer International Publishing book Fog Computing in the Internet of Things, 2017 [56]. Another article has been submitted to the IEEE Intelligent Transportation Systems Magazine; the response is still pending.

Chapter 5 concludes the thesis and highlights future research directions.

Chapter 6 presents the list of publications.

Chapter 2

iQ, a queue model simulation tool

2.1 Introduction

Computer architects use several simulation tools in order to design, evaluate and optimize computing systems. However, the wide variety of simulation tools makes the selection process a non-trivial task. While some techniques are based on analytical models [21, 51], others rely on the use of system simulators [13] for bottleneck identification and design verification. Popular software based simulators often emulate several different elements of a system including the processor micro-architecture, memory hierarchy, and interconnection network, but use different simulation techniques which impact accuracy, latency, ease of programmability, and analysis.

For instance, full system simulators such as Gem5 [13] are currently commonplace among researchers. Such tools allow simulating all the layers of the computer stack. Other researchers have been using trace-driven simulators [55] in an effort to reduce the execution time while maintaining accuracy. Furthermore, researchers use representative reduced traces [38, 76] which capture the workload behavior. All these aforementioned tools are excellent for detailed simulations and validating individual components. But they are cumbersome in dealing with the initial stages of design space exploration due to the lengthy simulation time and substantial development effort involved.

In contrast, there are other tools more suitable for design space exploration [16, 61]. They reduce the simulation time required while maintaining accuracy, but

the underlying complexity remains the same. Thus, another abstraction level is required. Analytical models should cover this area but are oversimplified. To improve the outputs of these models, some researchers [45, 69, 81] have used queuing theory to construct multi-threaded processors models to analyze resource contention without focusing on the processor implementation. However, finer granularity simulation is often desired when pinpointing micro-architectural bottlenecks or exploring diverse design parameters and components.

To meet this challenge, architects use a two-stage process. In the first stage, a high-level simulator is used for the design space exploration analysis. The bottleneck identification and the performance improvement estimation obtained guide the second stage, a more detailed simulation to test and validate component designs. In this work we present a fine-grained queue-based simulator, iQ, to be used in the first stage. The main requirements for iQ are a large complexity abstraction and fast simulations, while maintaining the error within acceptable boundaries. To satisfy these needs, we based our framework on queue theory and statistical information. The combination of these techniques allows us to represent any processor component or functionality with queues, servers, delays, and communication lines. While the queues correspond to the need of handling an instruction flow, the delays are the representation of the required time to perform an action over an instruction.

To represent applications, a dynamic instruction flow is generated based on a statistical profile formed by the instruction's distribution probability and register dependency information. Once the profile is available, it is time to build a processor model based on the queue elements. To construct processor models, architectural information is required, which can be determined easily for an existent processor (instruction width, ALUs, ROB length, etc) or in a new design the researcher defines these parameters. With the profile and the model, architects can study the impact of new components and/or analyze the bottlenecks on these models with simulations that take a few seconds, and most important modifications are feasible in real-time due to iQ's abstraction level. Later we demonstrate that accuracy is not lost to gain simulation speed.

Our goal in this Chapter is to develop a simulation methodology capable of performing efficient design space exploration analysis. Once such a tool is

available, we could evaluate the hardware and software requirements at the Fog nodes to identify the best architectures for IoT applications. To this end we analyze current simulation techniques and methodologies. After, we develop a generic framework based on queue models and statistical information. We build an illustrative example focusing on an Intel Ivy Bridge processor executing the SPEC CPU INT 2006. We then evaluate its accuracy and perform a design space exploration over this model to identify current bottlenecks and areas for future improvements.

In summary, the main contributions of this chapter are:

- We propose a simulation methodology based on queue models and statistical information for design space exploration and bottleneck identification at the core component level.
- We evaluate the speed, accuracy, abstraction level, and flexibility of the simulator. We validate it against real hardware and compare to other state of the art simulators.
- We demonstrate the usefulness of the simulator in its ability to provide efficient design space exploration and showcase available performance improvement options.

In Section 2.2 we provide a detailed description of iQ's characteristics and capabilities. Although our technique can be used to simulate any computer architecture (including processors, GPUs, and FPGAs), in Section 2.3 we detail how to implement a processor model, focusing on an Intel Ivy Bridge. In Section 2.5 we validate the Ivy Bridge model against real hardware and we also build and evaluate a Core 2 Duo model. In addition, we compare the simulation accuracy and speed of iQ with other state of the art simulators. In Section 2.6 we present a design space exploration analysis which showcases the usefulness iQ provides for architects and researchers, saving them vast amounts of design time and effort by quickly identifying bottlenecks and revealing improvement options. Finally, in Section 2.7 we review the related work and in Section 4.7 we conclude the simulation methodology.

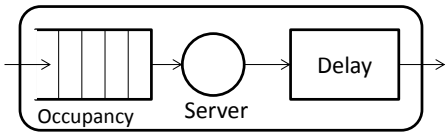

Figure 2.1: Generic modular queue structure. It is sequential and formed by three elements: a queue, a server, and a delay.

2.2 Background

Queuing models are based on queue structures, message passing, and latency accumulation to produce experimental results. A message is received at the tail of a queue structure, propagated until it reaches the head of the queue and then a delay is added to account for the amount of time that the action for that particular message or component is determined to take.

In the case of an arithmetic unit such as an Integer ALU in a processor, the queue can model the ALU input queue where the message represents an arithmetic instruction such as an add or subtract, and the delay added is the time the ALU unit takes to execute that arithmetic instruction. Dependencies between messages (i.e., instructions) or within computational resources (e.g., ALUs, branch predictors, Out-of-Order tracking) are also accounted to model the performance of the system characteristics being modeled.

A probabilistic model can also be included to emulate non-deterministic behavior such as branch miss-predictions and cache hits and misses. A collection of discrete events drive the execution simulation, in representation of computational cycles. Only the cycles where there is an event such as instruction generation, execution, or retirement are simulated, improving the velocity of iQ. Total performance is measured as a collection of processed messages per total events, in other words, instructions per cycle.

2.2.1 Queuing Model

Processors are formed by a wide variety of components such as functional units and different memory levels. To represent them using queuing model we have implemented a modular queue structure that is capable of representing different

Table 2.1: Generic queue structure configuration for different processor components

Processor component	Configuration
Non-pipelined Integer ALU	Server=ALU latency Delay=0
Pipelined Memory ALU	Server= 1 cycle Delay=mALU latency - 1
Partially pipelined Cache level	Server=non pipelined latency Delay=Cache latency-server

behaviors through a set of variable configurations. This module, represented in Figure 2.1 is formed by a queue, a server, and a delay. The users can configure the queue length and the delays required to process instructions. The **Queue occupancy** models the resource contention and availability.

Server: This parameter is used to model the time to execute the proper function over the instructions. The service time is the latency required to process an instruction. While an instruction is being serviced, the subsequent instructions wait in the queue. In other words, this parameter models how much pipelined a structure is. The lower the service time the higher pipeline the velocity and vice versa.

Delay: This parameter is used to complement the Server latency to ensure the appropriate total delay for the component the instruction will pass through. In this manner we ensure that the combination of Service and Delay time is used to represent any structure, pipelined or not, and attach the correct execution time to instructions as they are processed in the appropriate order.

In Table 2.1 we show some cases on how to configure the latencies to achieve the desired structure. For instance, if we are representing a non-pipelined structure, the total latency of an instruction processed by that structure will be entirely dependent upon the Service time and the Delay time is zero. A different case is a fully pipelined structure, such as Multiply ALU. Assuming that the total latency of this instruction is four cycles and a new instruction may begin execution each cycle, then the total execution time is the sum of a Service time (once cycle) and

a Delay time (three cycles).

Time-line: An event (i.e., cycle) is used to not only keep track of the number of cycles elapsed, but also tracks and schedules instruction events to maintain proper execution flow. For instance, assuming instruction A enters an ALU (which takes four cycles to compute) at cycle 42. Then, an action at cycle 46 is scheduled which will move instruction A to the following stage of execution. Performance is measured by dividing the total number of retired instructions by total elapsed cycles (i.e., IPC). The end of the simulation in reached when the variation between different IPC intervals is negligible and thus we consider the IPC is stabilized. The amount of time until the IPC reach that point is variable but usually is within tens of seconds.

2.3 iQ Methodology

2.3.1 Modeling hardware and software characteristics

Hardware: To properly construct such an accurate queuing model it is essential to understand the makeup of the target system's hardware components and the instruction mix of the workloads. Conceptually, architects need to have a high level view of a processor (like the 5-stage pipeline) to determine which basic modules are required to emulate the behavior of a processor. Ideally, we need at least three modules, one to create instructions, one to execute them, and one to retire them. Identifying the processor components to include in the model will determine the extra modules. For example, each ALU or memory level can be included with a generic module. To reduce the development effort and the simulation time, iQ models do not require knowing all the specific details but must only capture the main behavior of each component. For example, in constructing a cache module, details such as size, number of lines, replacement policy, and set-associativity do not need to be included in the cache module configuration. The cache module can still provide accurate results for processor models with only being configured to know the hit/miss ratios and corresponding latencies.

Software: To simulate instructions, iQ uses instruction types which are user defined classes resulting in a pseudo-ISA. This technique eliminates the complex-

ity and necessity of using binaries and compilers specific to our simulator. Applications must be profiled using hardware counters and tools like Pin [12]. This process enables architects to gauge the makeup of the application's instruction mix (e.g., arithmetic, memory, and branch) and register dependencies (distance between the creation and the use of value). This information can be detailed at the phase or basic block level to properly represent the different behaviors. iQ uses the profile to feed an instruction generator module that creates a representative code dynamically during the simulation. For instance, iQ uses a random number generator to produce different instructions types based on a probability distribution given by the application profile. As an example, if an application is composed of 75% load instructions, there is a 75% chance the instruction generator produces a load. The number of instructions generated per request is determined by the architectural parameter of the fetch and decode instructions per cycle, specified in the configuration file.

Accessibility : We use Omnet++ [73] to construct and simulate different hardware models. It provides an intuitive graphical interface to modify the modules conforming the processor's model, support for the libraries containing the generic queue modules or the user-defined functionalities. All the parameters that represent the software and the hardware are controlled via a configuration file. We provide a public release of the iQ simulator that can be used to develop new processor models.

2.4 Building an illustrative iQ model

2.4.1 Simulation setup

Target Architecture: We have used the iQ simulator to construct and simulate a model of the Intel(R) Core(TM) i7-3740QM CPU (Ivy Bridge). We evaluate a single core running single threaded applications. The architectural specifications for the Intel Ivy Bridge are publicly available [19, 24].

Host machine: We run our simulator on a Dell Latitude E6430 laptop. The processor is an Ivy Bridge with four cores and 8 GB of DDR3 RAM. On top of this platform, we have used Omnet++ 4.3.1 IDE to develop iQ models. Simulation

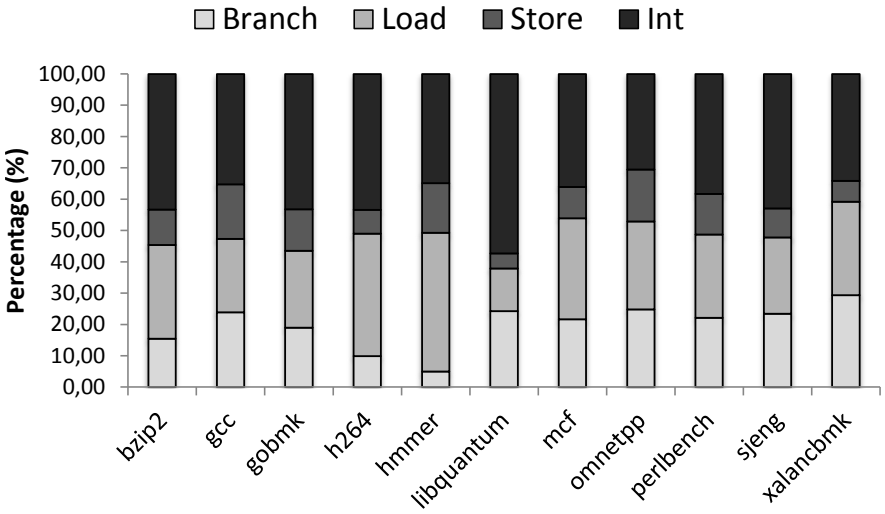

Figure 2.2: SPEC CPU INT 2006 Instruction mix divided into four types: integer, load, store, and branch.

accuracy and execution time are the two main characteristics evaluated. To provide a detailed and fair simulation evaluation, we compare our Ivy Bridge iQ model against the real processor. The parameter used in the accuracy comparison is the Instructions per Cycle (IPC).

Benchmarks: We evaluated our simulator running the SPEC CPU2006 Int benchmarks [31], except omnetpp benchmark since the dependency profiling tool was not capable of executing it and astar benchmark due to a segmentation fault in the Core 2 Duo. To obtain the application profile, we used the hardware counters via perf and Pin on a system OpenSuse 13.1 and gcc 4.8.11. Using the counters we can measure the instruction type distribution, the cache behavior, and the branch miss-prediction rate. We classify the different executable instructions to fall within one of four iQ's instruction class types which we have defined: Int, Load, Store, and Branch. We used the MICA tool [33] to obtain the register dependency distance between instructions.

Figure 2.2 presents the instruction mix falling into iQ's four instruction types for the SPEC workloads. This information is used by the instruction generator to determine the type of each new generated instruction following the same distribution as the original benchmark. Combining the hardware counters information with the instruction mix results can provide insight to determine the

critical processor's components for improving performance. Since iQ generates the instruction flow dynamically during simulation, the notion of a finite program vanishes into a non-ending execution. We consider that a simulation has finished once the IPC stabilizes.

2.4.2 iQ Ivy Bridge modules

Our processor's model structure is based on the 5-stage pipeline. Fetch, Control (joins Decode and Issue), Execution, Memory, and Retirement are modeled using iQ based modules detailed below.

2.4.2.1 Fetch

The fetch module represents the fetch stage and the L1 instruction cache (i-cache), plus the dynamic generation of instructions. An application can be composed of several phases with different profiles, and the architect can specify the phase execution order. In the SPEC CPU2006 Int case we observed almost a flat profile during the execution. In consequence, we defined a single phase profile information. The parameters required to categorize each phase include: (i) the distribution of different instruction types, shown in Figure 2.2 (ii) and the dependency information. The fetch module generates instructions based on this information.

An important parameter to represent the fetch stage accurately is the number of instructions per cycle that a real chip is capable of processing, which for the Ivy Bridge case is four [24]. Then, this module will use this information to generate four instructions on each request. We have not modeled the TLB since it is not critical for accuracy measurements when executing SPEC CPU2006 Int benchmarks.

Icache. Since we model the L1 i-cache, the simulator needs to determine whether a memory operation results in a cache hit or miss. This hit/miss ratio is set in the iQ configuration file. A random value is used to determine a hit or a miss according to the range obtained applying the miss probability to the desired distribution. In this case, we assume all the instruction misses go to the shared

L3 cache, and thus we apply the LLC latency (28 cycles [24]). If it is a hit, in the next cycle four instructions will be sent to the next module.

2.4.2.2 Control

This module is responsible for emulating the decode/issue stages and out-of-order execution, including processing of dependency checks and branch predictions. Instructions received from the fetch module are stored in the ready queue waiting to be processed. Similar to the fetch stage, the important parameter for modeling the decode stage is the number of decoded instructions per cycle that the processor can deliver. Before issuing an instruction to the modules emulating the execution stage, the control module must check the dependency information to determine whether the instruction will be blocked due to interdependencies or due to lack of free computational resources such as ALUs, Ld/St queue and ROB entries.

Modeling Dependencies. A consequence of representing the instructions with messages which do not include register information is that the register renaming and the pool of available registers have to be emulated with statistical information. To achieve this objective and also to collect insightful information, we use two queues. The first queue tracks the instructions under execution inside the processor. The second queue tracks the instructions blocked in this stage due to dependency reasons. To determine and control for inter-instruction dependencies, a dependency distance probability at the register level is utilized. Before issuing an instruction, a random number is generated which will determine if the instruction depends upon a previous instruction and what distance. The id of the instruction at the corresponding distance will be chosen as the one the current instruction depends upon. The current instruction becomes blocked until the instruction it depends on finishes execution.

Branch predictor. To predict branches, we use a similar method as with the instruction cache by generating a random number and checking whether it falls within the probability ranges of a true or false branch prediction. If the branch is correctly predicted, it is sent to the scheduler function which emulates the next stage in execution. On the other hand, if there is a miss-prediction,

the pipeline is flushed by emptying the ready instruction queue and a penalty is applied to the next clock event. This penalty sums the cost of the pipeline's flush and the average memory access latency to fetch new instructions. In the Ivy Bridge model the value is set up at 60 cycles.

Issue. The scheduler function checks if there is a free functional unit able to execute the instruction. In the Ivy Bridge case, up to five instructions can be executed simultaneously: three integer instructions (or 2 integer and 1 branch) and two memory instructions (2 loads, 1 load and 1 store, or one of either type). If there is an available functional unit (FU), the scheduler issues the instruction for execution by sending it to the corresponding FU module. In case the FU is occupied, it leaves the instruction in the ready queue to until the module becomes available. Out of order execution is simulated using the re-order buffer (ROB) length to define the number of instructions that the processor can examine inside the ready queue to find a suitable instruction to send. That length is reduced by taking into account the number of instructions under execution and also the blocked instructions. Once it finds an instruction, the model can send it even though it breaks the instruction sequence. The smaller length between the ready queue and the re-order buffer length defines how far this capability can go. Our ROB length is consistent with the Ivy Bridge architecture of 168 entries.

2.4.2.3 Integer/Branch functional units

These functional units are capable of executing three integer instructions or two integer instructions and one branch. To emulate their functionality the generic compound module from Table 2.1 is used. The required time to execute these instructions types in the real processor is one cycle so the service time is configured to be one and the delay value is set to zero. The queue length is unbounded because it is controlled through the maximum distance between the oldest instruction and the one to be sent.

2.4.2.4 Memory hierarchy

The Intel Ivy Bridge processor is capable of executing two loads simultaneously, represented with two memory FUs. The cache miss ratios are specified for each

memory level. iQ uses a random number to determine the outcome of the memory accesses. To execute the operation, a cache control module is required after the generic functional unit. After the memory instruction goes through the L1 d-cache modules, it arrives to the L1 control module. At this point, whether the memory access is a cache hit or miss is determined. If it is a hit, then the instruction is sent to the retirement module. In case of a miss, it is sent to the L2 cache module. The same procedure is applied for L2 and L3. Main memory accesses are treated differently because they always hit.

2.4.2.5 Retirement

The retirement module emulates the retirement stage of a processor. The processor model retires instructions in an out-of-order fashion since instructions are retired when they arrive. This does not affect the accuracy since iQ uses statistical profiles of the application and not real code. If a traditional trace was used instead, then the in-order retirement would have to be used. Once the instruction is retired, its instruction identification number is sent to the control module. The control module can then check possible dependencies on that id and proceed to execute those instructions. This module also collects statistics about the entire simulation such as the latency required to execute each instruction, histograms about queue occupancies, number of retired instructions separated by type, etc.

2.4.3 iQ Ivy Bridge model

Figure 2.3 presents the model once all the aforementioned modules are combined.

2.5 iQ Performance Analysis

Figure 2.4 presents the comparison between the simulated and real Ivy Bridge IPC values running the SPEC CPU2006 Int benchmarks. It also shows the percentage relative error between these values. Apart from gcc that will be explained separately, the average error rate for the other benchmarks is 8.6%. This error is comparable to those obtained with cycle-accurate simulators specifically modified to match a specific platform [29].

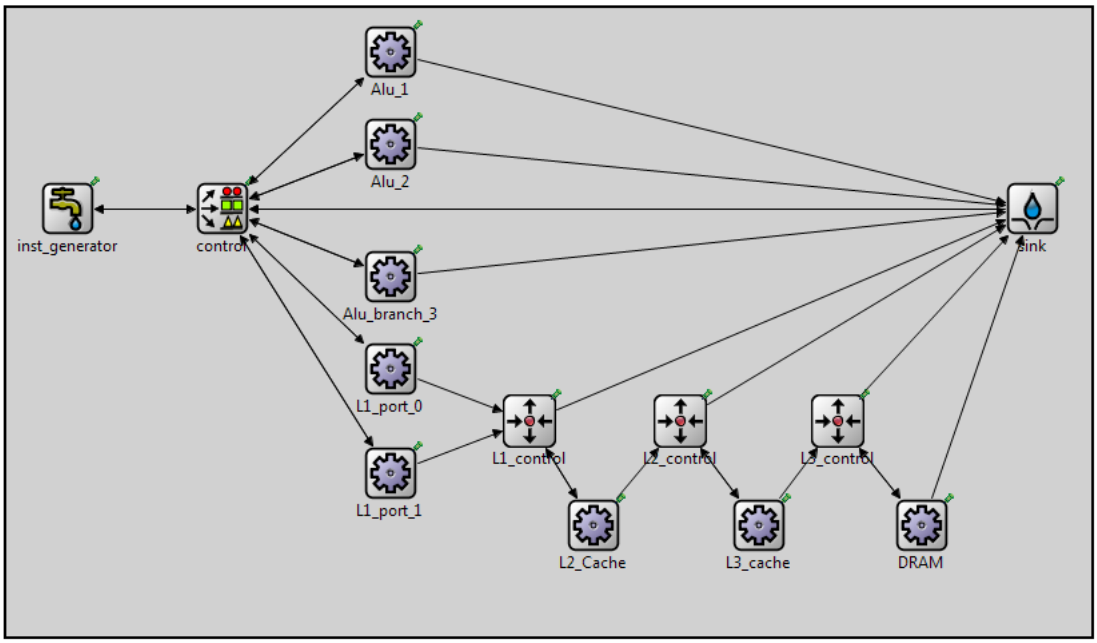

Figure 2.3: Intel Ivy Bridge model implemented in Omnet++

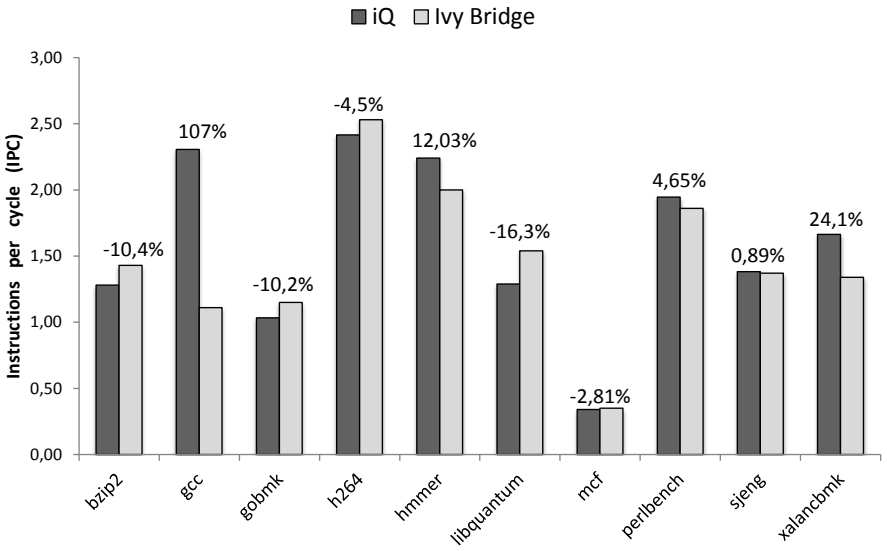

Figure 2.4: IPC comparison between iQ model and real Ivy Bridge with error rates shown on top. The absolute average error rate (except gcc) is 10.5%

Figure 2.4 illustrates that the model is able to accurately execute a wide set of benchmarks (both memory and computationally intensive). Moreover, the

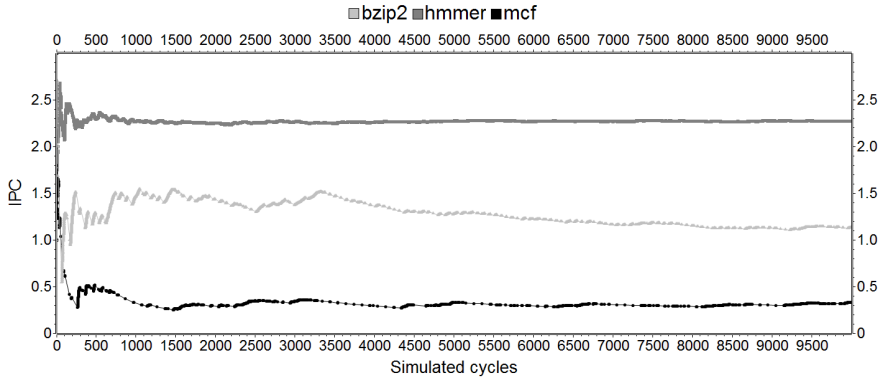

Figure 2.5: IPC evolution during simulation. After less than ten thousand simulated cycles, which take a few seconds to execute, the IPC stabilizes.

achieved accuracy proves that we emulated reasonably the key processor's components. There is only one outlier, gcc. After analyzing its code, gcc can be represented as small phases with completely different characteristics and instructions. Reason why our four instruction generator with a single phase at this stage cannot represent the application code accurately, provoking the error observed. As future work, a more detailed phase representation has to be implemented to allow users to simulate this type of benchmarks.

2.5.1 Simulation Speed

The other important critical characteristic of our model is its fast simulation speed. iQ's execution time is reduced by the fact that the model only needs to execute the application profile until the variation in the output IPC value is negligible and does not need to execute the entire program. Figure 2.5 presents how the IPC of three benchmarks executed in the Ivy Bridge model evolves based on simulation cycles. In less than ten thousand cycles all the benchmarks have a stable output IPC, and thus the simulations can finish. After measuring the real CPU execution time required for simulating those cycles, the final IPC is obtained in 2 seconds on average with a maximum of 4.2, demonstrating a remarkable speedup over the times required by other simulators.

Table 2.2: Comparison between iQ and other simulators

Simulator	**Avg. error (%)**	**Avg. Sim time** (hours)
iQ	8.6	0,0005
Analytical [70]	13	0.055
ZSim [61]	9.7	1,12
Sniper [16]	19.8	6,94
MARSS [52]	15.5	86,80
Gem 5 [29]	13	69,4

2.5.2 Comparison with other simulators

Table 2.2 compares the absolute average error (second column) and average simulation time (third column) between our iQ simulation platform and other state of the art proposals. To populate it, we used the numbers from the original papers. The third column presents the execution time for each simulation technique: Gem5 runs the test input set of the SPEC2006, Sniper runs the large set of Splash-2 [75], ZSim runs 50 billion instructions, analytical models run 1 billion and iQ runs until the IPC is stable. We see that iQ is the fastest and provides very low error percentage.

Gem5 has been the "de facto" full system simulator to test and evaluate processor components. Its simulation detail results in very slow but highly accurate simulations. However, the development cost is high. For example, a customized version of an ARM processor compared to real hardware obtained and average absolute error of 13%, while the simulation time is in the range of hours per benchmark [29]. MARSS presents similar results in terms of accuracy and simulation speed. Both simulators are challenging for design space exploration due to the development effort required for modifications and for the long execution time.

Sniper's scope is to perform rapid and accurate simulations by using interval simulation. Sniper has been validated against a real x86 processor executing the Splash-2 benchmarks. The simulation speed of Sniper is improved compared to full-system simulators like Gem5 and MARSS, in the range of a few MIPS, while

the accuracy is relatively high. However, the flexibility is still limited because performing architectural changes takes time and the simulation latency can still take hours.

ZSim aims to solve this issue among others such as the scalability. It is based on instrumented code obtained from Pin. ZSim achieves better simulation speeds than tools that simulate in detail almost all the processor's components and provides excellent accuracy. It is a suitable tool to perform accurate simulations, but not to perform design space exploration analysis because the execution time is not small enough and modifications still require significant efforts for such high-level studies.

Analytical models present similar simulation speeds to iQ but the resulting accuracy depends on the model's complexity. In Table 2.2 we show one of the latest models [70]. To achieve that accuracy, that model is based on interval simulation and complex equations. Furthermore, it implements cache and branch predictors which are complex for such a model. It achieves reduced execution times using checkpoints. However, analytical models provide average performance values which obscure the dynamic behavior. This information is crucial for performing optimal design space exploration analysis and modifications to the processor architecture itself (more ALUs, memory ports, etc).

Conversely, an iQ based processor model does not implement all the processor's components. The desired components emulate the real behavior through abstractions to simplify the simulation framework. It is not capable of executing the operating system and instead of using the actual binaries of the applications, it dynamically generates a trace based on statistical profiles. Despite all these facts, the accuracy level outperforms nearly all other simulators and simulation time is better than that of complex analytical models. In addition, it provides detailed simulation information such as dynamic instruction and component queue behaviors. Thanks to the component abstractions and the instruction generator, iQ performs simulations within a few seconds compared to several hours. These benefits make iQ a suitable tool for design space exploration analysis and processor design.

2.6 Design Space Exploration Analysis

iQ has been developed to perform architecture analysis and bottleneck identification for optimizing existing architectures and identifying potential for new architecture designs. The advantages iQ provides over other simulators are the ability to accurately identify architectural bottlenecks and speedily simulate a wide range of design modifications through different abstraction levels. Experimenting with modifying subtle design parameters such as to add new components such as functional units can often be a daunting task both in coding effort and simulation time required with current simulators. The ease and speed which iQ provides allows architects to try out large quantities of design space studies and produce optimization options featuring different combinations of component modifications. This section demonstrates the practical usefulness of the iQ model and how it can be used by architects to run design level analyses and optimize their systems.

The Problem: Imagine a scenario where a computer architect has to improve the Ivy Bridge processor by 15% for the SPEC CPU2006 Int benchmarks (except gcc, omnetpp, and astar) within a week. The architect decides to use the new iQ model and set aside other simulators that are either more complex or that lack enough abstraction level. Different steps will lead the architect to that goal.

The first step is to profile the benchmarks, build a high-level model of the processor, and validate it. This is achieved following the methodology described in Section 2.3 or using predefined modules that should be available for commercial processors. The next steps to determine the architectural bottlenecks limiting performance are discussed in the following Sections.

2.6.1 Multiple Parameter Analysis

Though a single parameter analysis could provide the desired improvement, it ignores the correlation between architectural factors. For example, how does the performance vary as both the fetch width and the number of functional units are modified at the same time? This is where iQ's speed is specially useful.

Architects can perform multiple parameter analysis thanks to executing a large number of simulations within a reasonable amount of time (i.e., orders of

magnitude less than with full system simulators). Additionally, iQ's abstraction level offers an advantage over analytical models since the underlying complexity is reduced and the dynamic behavior of the processor's components (ROB, fetch queue, etc.) can be analyzed. Now, modifying the parameters of each factor is practically effortless since it only requires changing one parameter variable in the configuration file per factor being measured (e.g., change the OoO window to 128 and the L2 cache hit rate to 90%).

OoO and Fetch width: Based on the single parameter analysis performed before, the architect decides to evaluate the correlation between the out of order (OoO) window size and the fetch width. Figure 2.6 represents the results between these two factors. The architect decided on examining this relationship by modifying each factor using nine different parameter values. As a result, the number of required simulations per benchmark is 9*9=81, which for all 9 benchmarks means conducting a total of 729 different simulations. Using iQ, all 729 simulations took less than 25 minutes to run on the laptop described in Section 2.4.1. The amount of time required is smaller than with any other simulation technique while iQ's accuracy stands with much more detailed models. Additionally, this correlation study can be applied to analyze the relation between any architectural factor configured in the model which greatly increases the already large number of simulations needed.

Several conclusions arise from the information presented in Figure 2.6. For example, increasing the number of fetch and decode instructions does not provide increasing returns if the out-of-order window length remains relatively small. The extra instructions fetched will not fit within the provided OoO window resources to execute them simultaneously. Similarly, if the fetch width remains relatively small but the OoO window is greatly increased, performance does not increase since not enough instructions are fetched to keep up with the larger OoO window size. Maximizing the value for both parameters can increase performance but its implementation will be impractical and a waste of resources since more conservative configurations produce similar results. A leveling off of performance gains indicates that the performance is becoming limited by bottlenecks besides these

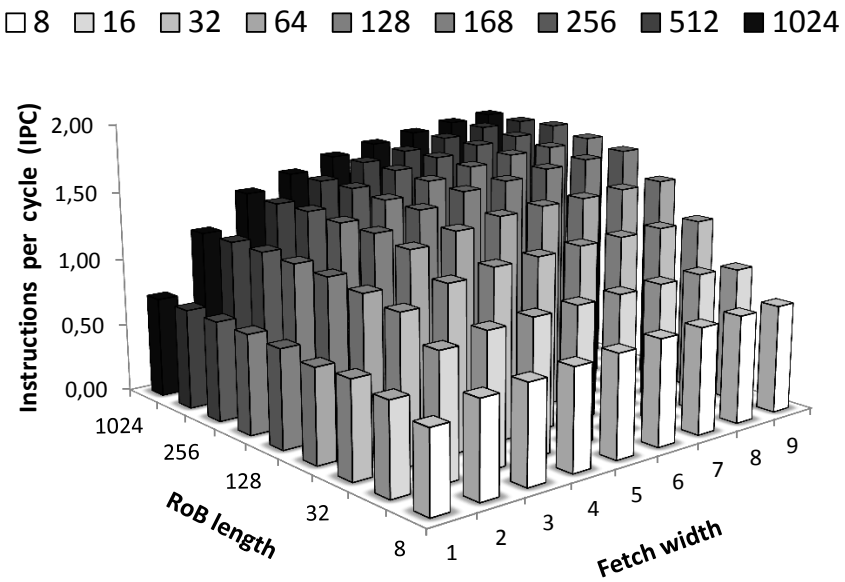

Figure 2.6: Analysis between OoO window size (or RoB length) and fetch width. Increasing the fetch width over the 168 (Ivy Bridge default value) to exploit larger OoO window lengths does not result in significant IPC improvements.

two factors. The optimal configuration, in terms of IPC and implementation feasibility, is having a OoO window size of 128 and a fetch and decode width of 6 instructions per cycle.

Fetch width and branch penalty: Note that previous analysis showed that the OoO window size is too aggressive for these benchmarks but the fetch width is important. Hence, the architect decides to analyze the correlation between the fetch width and the branch miss-prediction penalty. Figure 2.7 presents the results of this analysis.

The first thing the architect notices is that as the fetch width increases over five instructions per cycle, it does not provide a significant IPC improvement. IPC performance, however, becomes heavily dependent upon the branch miss-prediction penalty once the optimum fetch width value is attained. This is in stark contrast to what was observed in the relationship between the fetch width and the out-of-order window length where after a length of 128 the improvement was negligible. This fact highlights that not all factors are equal nor have equal relationships with one another. Different factors can limit or enhance the poten-

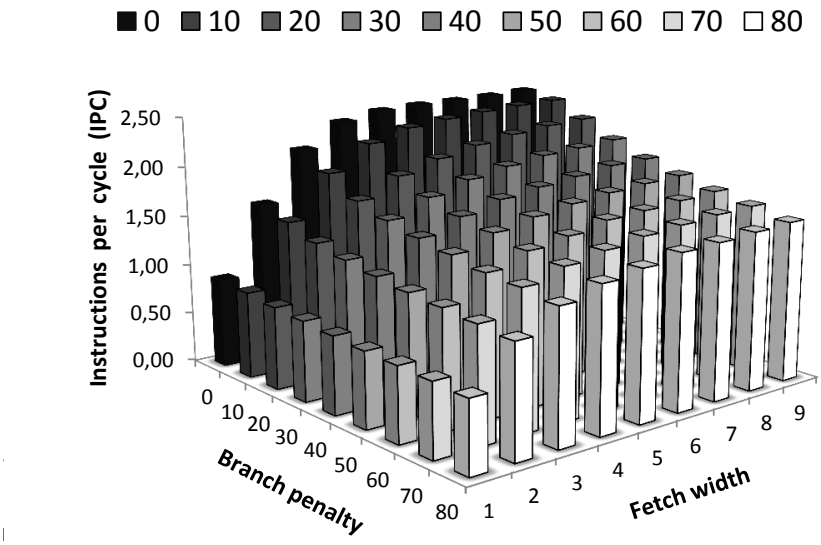

Figure 2.7: Analysis between fetch width and branch misspredict penalty. There is a strong correlation between the branch penalty and the fetch width observed in the non-saturated IPC improvement.

tial benefits of others. The particulars of these relationships are determined by the characteristics of the architecture and applications.

2.6.2 Complete Analysis

A modern processor is a complex machine with many more factors and parameter values that should also be evaluated. This fact indicates that the number of simulations growths exponentially with the number of factors. Based on the insights from previous analysis, the architect decides now to use the factors from the multiple parameter study (shown in Figures 2.6, 2.7) and the LLC and DRAM latencies from the single parameter exploration. Conducting an experiment consisting of 7 different parameter values for these 5 different factors concurrently means 16807 simulations for each benchmark (9 hours to run).

Eight configurations out of this multi-factor study are presented in Table 2.3 and compared to the default Ivy Bridge architecture. They form a representative subset of solutions based on the number of factors involved and the improvement

achieved for comparison reasons. Although there are more configurations that accomplish the objective, choosing the appropriate solution will depend on the cost functions accounting for power, area, and implementation feasibility.

Regarding the results from Table 2.3 the only difference between configuration A and configuration B is the increase of the size of the RoB length in B to 256. A larger number of fetched instructions per cycle combined with a reduced branch penalty makes that more instructions are available for execution each cycle. Then, this fact is exploited by a larger RoB length which translates into an improvement of 18.19%, while A obtains 15.33%. It becomes the architect's job to make a cost-benefit analysis and determine whether this extra 2.86% improvement is worth the implementation and energy costs. Configuration G provides a similar improvement, a 17.03%, with a different combination of parameter values. Instead of modifying the RoB length and the branch penalty it reduces the LLC miss ratio by 5% and also reduces the DRAM latency. Improving the memory hierarchy reduces the time instructions are blocked due to pending memory requests and allows to exploit a larger instruction level parallelism (ILP).

Following with the memory improvements, configuration C includes a reduction of the LLC miss ratio by 10%. In this case, a less aggressive fetch is compensated with a reduced branch penalty. However, this configuration is not so optimal as the previous one and presents a 15.46% improvement. Knowing that the LLC cache miss ratio needs to be decreased the architect can then use a cache simulator to decide which cache scheme fulfills the new miss ratios. Also remarkable is the configuration H, where all the factors are slightly modified. In comparison with the previous configurations where specific components were targeted, H proves that a minimum enhancement in all the processor stages achieves the desired 15% improvement.

Other solutions are more aggressive, such as E. To achieve a drastic reduction of the DRAM latency may not seem feasible. However, new memory technologies may enable such breakthroughs and then the architect can estimate its impact. Once configuration E revealed its potential with a 15.32% improvement, the architect can iterate on top of it. Decreasing the RoB length following the premises from previous analysis to 128 and reducing the branch penalty by 10 cycles results in 16.55% improvement (configuration D).

Table 2.3: Design space exploration analysis: factor and parameter configurations with the resultant performance improvement

Configuration	**Fetch** (inst/cycle)	**RoB length** (entries)	**Branch penalty** (cycles)	**DRAM latency** (cycles)	**LLC latency** (cycles)	**Improvement** (%)
Default(-)	4	168	60	180	28	-
A	5	128	40	-	-	15.33
B	5	256	40	-	-	18.19
C	5	-	50	-	Miss ratio -10%	15.46
D	5	128	50	130	-	16.55
E	5	256	-	125	-	15.32
F	-	256	40	150	-	15.28
G	6	-	-	140	Miss ratio -5%	17.03
H	6	256	55	170	25	15.00

2.7 Related Work

Different techniques have been used to provide feasible design space exploration tools. SMARTS [77] provides reduced representative subsets of benchmarks to reduce the simulation time although the underlying processor model can compromise its advantages. To avoid the use of third tools, synthetic traces [51] can be generated recreating the original behavior from a previous execution reducing the simulation time. TaskPoint [28] applies sampled simulation to task-based programs. Cook et al. [18] developed a design space exploration technique based on Monte Carlo methods. Lee et al. [39] used a regression model to analyze the trade-offs between performance and power consumption.

Prior work has also used queue models to simulate multiprocessor systems [68, 69, 81]. They exhibit a higher level of abstraction in the processor architecture, reduced to a traffic generator, because they focus on the multi-threaded contention problems. The first work only simulates the different cache levels of the memory hierarchy and how the requests access them. They do not simulate the ISA, focusing on the memory accesses. The second work adds more detail to the memory hierarchy implementing the bank scheme and the main memory, but the rest of the processor still remains hidden. iQ extends both works by implementing the remaining processor components, a more complete ISA, and defining a generic framework easily extensible to other processor architectures. Then, iQ can perform a fine-grain analysis of the entire processor.

Data center scale simulations have been performed using the queue methodology, both in performance [46] and in power [45], sharing the granularity problem of previous works.

2.8 Summary

In this Chapter we proposed iQ, a tool primarily focused on helping to perform design space exploration analysis and identifying bottlenecks. However, iQ does not provide answers to how a specific parameter configuration may be achieved at the implementation level. For instance, while improving the hit ratio of a cache to a certain feasible percentage will help to optimize the system, iQ does not identify

what the specific cache structures (e.g., set-associativity, line size, etc.) should be to achieve that goal. To this end, a detailed simulator should be used. iQ should be complemented with detailed system simulators for validating detailed physical implementation designs. However, substituting detailed simulators for iQ in the design space exploration phase results in a significantly faster and more insightful processor design and evaluation process for computer architects.

Chapter 3

Fog Computing, towards a generic platform for IoT

3.1 Introduction

Based on estimations, in 2020 there will be around 50 billion devices connected to the Internet [22]. These "things" are in their majority sensors and actuators interacting with the real world. Hand by hand with these devices, a huge amount of data is collected, requiring a process of analysis and actuation. These applications must carefully assess certain crucial factors such as the real-time and largely distributed nature of the "things", maintaining trustworthy communications, and adapting for mobility and the harsh environments where the "things" are deployed. To achieve these objectives with traditional Cloud Computing solutions is complicated since a centric design approach precludes critical IoT requirements, such as real-time and geographically-aware computing.

Fog Computing [15], an architecture that appeared in the past years, can be used as a base to provide a good solution to the IoT requirements [78]. It is a highly distributed platform, with nodes located from near the end-user devices till the edge of the network. These nodes offer resources such as computing, storage, and networking to the applications operating under this infrastructure. An access point with enhanced computing capabilities constitutes an example of a possible Fog-capable node. Fog processes the data close to where is generated, reducing the network utilization and improving the aggregation from the bottom

of the infrastructure [66]. Low-latency, wide-spread geographic distribution, heterogeneity, and mobility are part of its main advantages. In consequence, Fog becomes an extension of the Cloud rather than a substitute since its nodes are connected to the Cloud.

The true potential of Fog Computing lies in the implementation of a generic multi-tenant platform supporting a wide rage of applications simultaneously [5]. This approach reduces deployment costs, eliminates hardware redundancy, and improves the scalability of the system. However, current IoT deployments are based on "things" covering an area with a set of proprietary nodes connected to the Internet. As a consequence, each application constitutes a subsystem or silo [79] inside IoT and there is no exchange of information among applications that users could benefit from. Even though a layer of Fog nodes is available, applications have to develop complex software solutions to integrate those nodes into their infrastructures or face vertical deployments from the "things" to the Cloud. Then, a paradigm shift as shown in Figure 3.1 is required to enable a generic IoT infrastructure.

For Fog Computing to grow as a platform and reach that vision, it should adopt the ease of deployment from on-demand platforms such as Cloud and the flexibility from software defined technologies such as SDN. To accomplish these

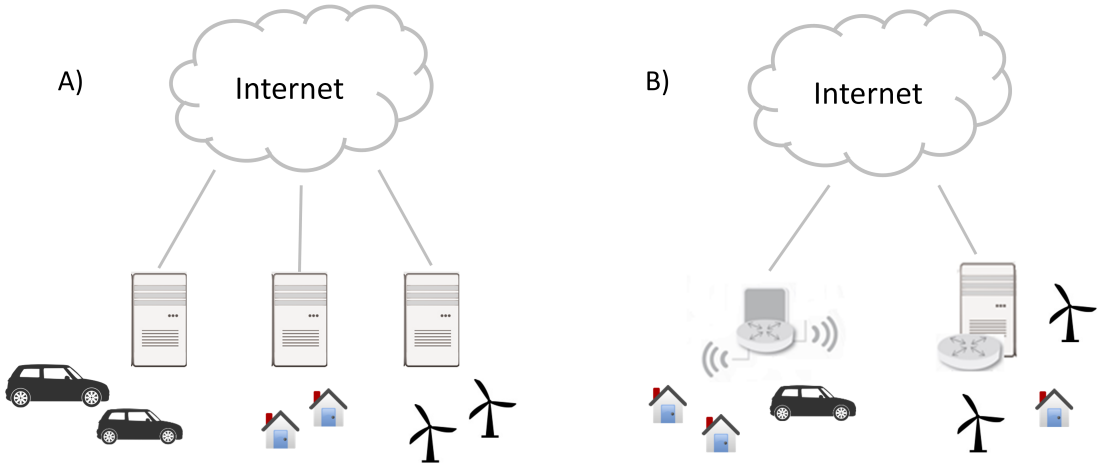

Figure 3.1: Moving from a silo-based implementation where each application has its own infrastructure represented in (A) to a generic Fog deployment capable of executing several applications simultaneously represented in (B)

objectives, we propose to enhance Fog with three key innovations. First, a new orchestration policy to provide more flexibility to the infrastructure breaking the execution in the Cloud by default. Second, the creation of constellations of heterogeneous Fog nodes to aggregate their capabilities, increasing the available resources at the bottom of the hierarchy. Last, the definition of the Fog Function Virtualization (FFV) concept. These functions cover aspects such as analytics, sensors' functionalities, and computational resources among others. Hence, the system exposes its capabilities through these functions without jeopardizing the complexity nor the scalability.

The combination of these three techniques contributes to the democratization of the IoT services by truly enabling a generic infrastructure to run multiple applications simultaneously. The deployment of a new service only requires the application code without worrying about the node integration, since virtualization techniques hide the entire infrastructure from "things" to Cloud.

Our goal in this Chapter is to enhance Fog Computing to become a generic platform capable of executing a heterogeneous set of IoT applications simultaneously. Once such a platform is available, many IoT systems can be deployed easily since the entry barriers such as the complex software development and the necessity of owning entire verticals will be eliminated. Later, these systems will become the basic units to enable a true IoT ULSS. To this end we analyze current Fog Computing architectures and deployments to identify the critical areas that are precluding the expansion of Fog-based applications. After their identification, we design we three enhancements to attack each of the problems and analyze their applicability in different scenarios.

In summary, the main contributions of this chapter are:

- We identify three critical areas that are precluding the expansion of Fog-based deployments: (i) orchestration policies, (ii) resources available at the first hierarchical levels and, (iii) the ease of infrastructure resource instantiation

- We propose three enhancements to alleviate the aforementioned problems: (i) a new orchestration policy, (ii) the creation of constellations of nodes and, (iii) the definition of FFVs.

- We demonstrate its usefulness in two different scenarios.

In Section 3.2 we first describe the Fog architecture, its functionalities and the desired requirements. We then elaborate the three innovations proposed in Section 3.3. Section 3.4 illustrates its usefulness in dealing with different IoT deployments through a two scenario case study. Finally, Section 4.7 summarizes our analysis.

3.2 Fog Computer Architecture

A representative architecture of a generic IoT infrastructure is depicted in Figure 3.2. At the lower levels there are "things", responsible of gathering information. The next layer is formed by heterogeneous Fog nodes, which constitute the aggregation points. The "things" and the nodes communicate mostly through wireless technologies, since both "things" and nodes can move. In addition to this vertical communication, Fog nodes can communicate horizontally (i.e. between two Fog nodes at the same hierarchical level). Due to Fog nodes' wide geographic deployment and their locality, they can offer real-time resources processing the data close to where it is generated. These characteristics enable most of the Fog Computing advantages (i.e. mobility support, low-latency, etc.). Till reaching the Cloud, Fog nodes form an interconnected hierarchy. Among these nodes there might be non-compatible Fog devices operating normally. The Cloud constitutes the last layer, offering a large pool of resources at low-cost but without any latency guarantees.

The decision on which layer executes the application depends on its requirements although the final decision corresponds to the orchestrator. The original Fog architecture defined that applications run on the Cloud by default and only those which strictly require Fog capabilities use the Fog layers. Once the Fog layers are chosen, other parameters such as the node's visibility serve to decide which Fog nodes execute it. Exploiting the visibility is important for applications covering a wide geographic area, since aggregation can take place at higher nodes. The resultant advantage is a traffic reduction since just the strictly necessary data is sent from one level to another.

Now, the objective turns to optimize the applications' execution. To this end, both Cloud and Fog use virtualization techniques. They rely on virtual machines to offer security, isolation, dynamism, and ease of management [35]. Recently, the "things" themselves are offered as a service too [9] [53]. For example, a company deploy a set of complex sensors measuring different events. They can offer virtual instances of these sensors to other companies fulfilling the requirements of a new application. In consequence, companies have at their disposal the entire infrastructure as a service.

Security and privacy pose many challenges that delay the IoT explosion [10]. While silos provide some natural protection mechanisms due to its isolation, a multi-layer infrastructure augments the attack surface. In addition, Fog operating as a generic infrastructure poses new threats such as side channel and resource exhaustion attacks. For instance, an attacker could access the "things" themselves and make an IoT device to malfunction. In certain cases such as a pacemaker this is critical.

Furthermore, people are reluctant to expose personal data in the Internet arising from a wide range of IoT devices. And aggravating the problem, different legislations apply and there is no consensus on security or privacy standards. In any case the protocols implemented cannot jeopardize the real-time characteristic and should be able to run on the low-power simple devices in which IoT relies on.

3.2.1 Implementation requirements

We have seen the basics of Fog architecture and how it deals with the new requirements brought by IoT. The hardware platform required to deploy new applications is available, but not so with the software. To integrate these Fog devices into a current infrastructure requires a vast effort to develop the algorithms that manage the execution through the different layers. This fact has prevented the explosion of Fog-based deployments. Then, the pending question is: What are the desired characteristics from the application's perspective to make Fog an attractive and successful solution?

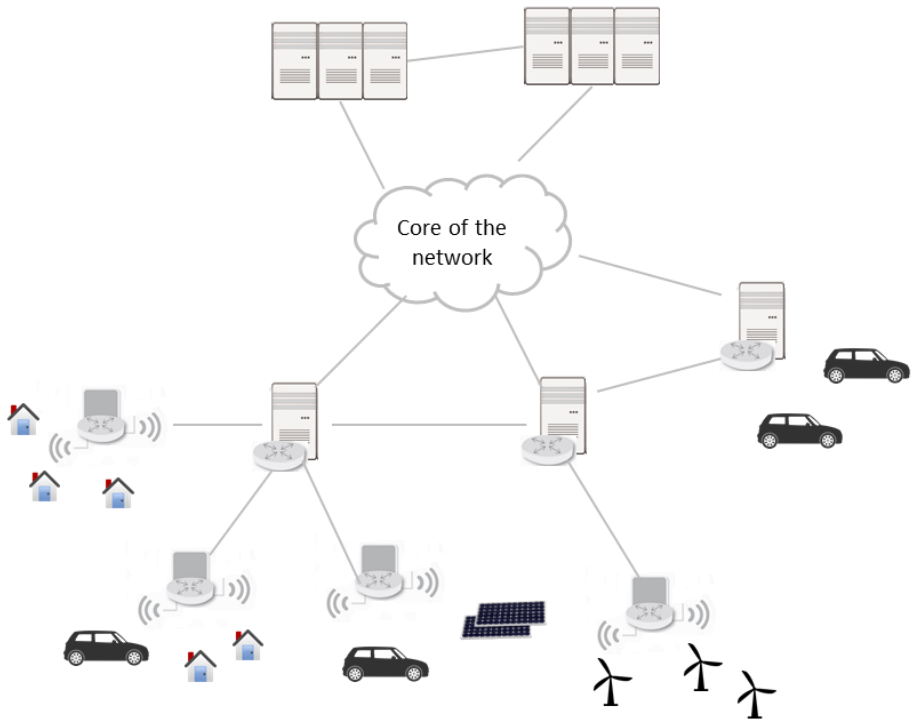

Figure 3.2: Illustrative example of a generic Fog-based infrastructure serving multiple IoT applications. Fog nodes are interconnected forming a hierarchy.

The system should present a great flexibility to offer its hardware resources. Regarding this aspect, the on-demand based Cloud solutions have proven themselves as an optimal technique [35]. New users can ask for instances to start running their applications within minutes. The same solution can be applied to Fog. Now, these instances cover the Fog nodes that bring extra complexity. The nodes can belong to different companies, they can have different capabilities, and they are distributed geographically among others. Hong et al. used the assumption of ideal instances to define their API [32]. However, they exposed the entire infrastructure to the application developers, aggravating the software problem. Since mobility is an IoT pillar, it is really difficult to anticipate which devices are in a certain area to program them in advance.

To make a successful IoT platform, applications would like those instances to

be totally transparent to them. They do not want to know the nodes responsible of executing their code or who owns them as long as their requirements are satisfied. Among these requirements security is critical. If security is weak, there is a potential lack of control over the data. Additionally, users do not expose sensitive data without certain guarantees that today are not fulfilled. In consequence, IoT applications will not achieve the market estimations unless these issues are solved.

Another critical aspect in these instances is the connectivity between the different layers. First, applications use a wide range of protocols to communicate "things" with Fog nodes. However, this diversity may suppose a problem if different hardware is required. For instance, a Fog node could execute two applications, one using Bluetooth and the other using WiFi. In this specific case, that node has hardware support for both standards. Once the information is in the Fog layers, the network among these nodes is not uniform and this fact may affect the application requirements. Hence, applications need adaptivity and transparency regarding the interconnection system, having the generic platform for IoT as the final objective.

The aforementioned requirements deal with a key aspect, the ease of deployment. If new applications just require an instance to start running their code without the concern of managing the entire infrastructure, Fog will become the "de facto" IoT platform.

3.3 Innovations beyond Fog

To facilitate the achievement of the goal presented in Section 3.2.1, we present three key innovations to Fog. Each technique applies to a different area and their combination enhances the flexibility of the platform to resemble that of Cloud. In addition, our solution extends prior work on Cloud and Grid Computing to adapt to real time constraints and latency requirements imposed by the "things" and the critical applications they enable [25]. The areas where we focus are the orchestrator, the resources available at the Fog layers, and how the infrastructure offers its capabilities to the applications.

3.3.1 Distributed orchestrator

The first step to implement that flexibility and overcome its limitations [74] is the modification of the orchestration policy that englobes the policy decision and the policy enforcement point. It is necessary to break the default Cloud option to exploit the advantages of the new Fog layers and their capabilities. At the end, Fog Computing implies the location of computational resources close to the "things". Then, our proposal is to execute the applications on the most convenient layer without a preset decision.

If the Fog layers are selected to execute workloads, the orchestrator takes into account different parameters to decide which Fog nodes are the responsible for each task. These parameters include geographic proximity, congestion, node's capabilities, and application requirements. Precisely, these two last parameters drive the decision. If a node does not have the required resources, it is automatically discarded. For example, if a node cannot guarantee a certain latency response, critical applications must discard that device.

Once this matching process is solved, an equitable distribution between the different nodes becomes fundamental. The system needs free resources to allocate the dynamic IoT applications while the execution of static services continues. In parallel, it can exploit the node visibility to its advantage. If an application executes under a unique Fog node, there is no need to migrate it to higher nodes. For example, imagine a wide range of sensors deployed within a smart building. In this case the closest Fog node can process all that data keeping the applications running. Only some statistical information can be sent up in the hierarchy if required.

All together allows to exploit the advantages of the Fog layers, optimizing executions over the infrastructure. Now, data is processed close to where it is generated and consumed. Consequently, aggregation takes place at lower layers of the hierarchy, eliminating unnecessary traffic and reducing the bandwidth because only necessary data is sent to the higher layers. Although bandwidth is not a problem yet, transmitting data from billions of "things" to the Cloud may pose structural problems to the underlying infrastructure.

3.3.2 Constellations of Fog nodes

The orchestrator modification led to another problem, the resources available at the Fog level are not the same as at the Cloud. The Fog nodes are a compendium of heterogeneous devices that are geographically distributed. Some of these nodes are complex devices with many capabilities (i.e. server with enhanced communication capabilities) while others are pretty simple (i.e. gateway) [80]. Hence, the infrastructure may not have the required capabilities in a desired location to execute a service [71].

To avoid sending these workloads to the Cloud or to unnecessarily deploy more devices, we propose to create constellations of Fog nodes. Aggregating the nodes' capabilities gives the perception of larger pools of resources close to the end users, although still far from the Cloud resources. These larger sets of resources come at a cost of latency since they rely on distributed nodes. Here there is a trade-off between the constellation latency and the Cloud one. If Fog nodes exploit their locality to create these groupings, their latency should be below that of the Cloud. However, in certain situations where further nodes are needed the Cloud may appear as a feasible solution.

The idea of sharing resources was also presented in other environments such as Mobile Cloud [49]. They combine user devices capabilities without including system nodes in their local clouds, reducing its advantages. In opposition, constellations focus on Fog nodes from different layers and virtualization becomes the way to create and offer them.

Through virtualization, constellations also eliminate the view of multiple owners. Applications observe capabilities on a per constellation basis and not for each node individually. These virtual groupings also enhance the security and ensure the isolation between applications running on the same Fog nodes. The criteria to form groupings can be proximity or to add a certain capability such as hardware accelerators. Once groupings are formed, constellations can prioritize determined user demands based on the criticality of the application. Furthermore, the Fog nodes forming a constellation change dynamically due to the mobility of the devices. For example, a node on a bus can join/leave constellations based on the vehicle mobility.

Once the infrastructure uses this technique, applications have the required resources close to the "things". Workloads can be executed close to where the data is generated and thus exploiting the main Fog advantages without deploying extra hardware.

3.3.3 Fog Function Virtualization (FFV)

Currently, the deployment of a new service is a daunting task. Within the complex software required, there are the functionalities to exploit the devices and their capabilities. Solutions like Service Oriented Architectures (SOA) also face the same problem due to lack of abstraction in the devices' functionalities [20]. Then, the subsequent allocation between applications requirements and infrastructure capabilities becomes critical to enable IoT services.

Nowadays, solutions rely on proprietary code designed for specific devices that prevents reutilization. This fact supposes a barrier for new applications due to the huge effort required to deploy a new service and delays the explosion of Fog-based applications. Building on the concept of Network Function Virtualization (NFV) [17], we applied it to IoT and Fog computing through the definition the concept of Fog Function Virtualization (FFV). With these functions, a Fog-based architecture exposes their capabilities as high-level characteristics, regardless of each application aims. Computation and storage can be controlled as functions, but also "things" functionalities and analytics.

FFVs map the applications' requirements into the capabilities of Fog nodes and "things". As a result, the application development is reduced to choose the FFVs that produce the desired service. To achieve complex functionalities, developers may choose a chain of functions. In order to increase functions' re-usability, preexisting FFVs can be available through libraries. Each of these functions has a set of inputs (i.e. sensor measurements), it performs a processing, and produce an output (i.e. their average value) to be used for other application's layers.

Since Fog nodes are highly heterogeneous, FFVs are not capable of running in all the Fog nodes due to the nodes' capabilities. In case a function cannot run in the designated node there are different options. One of these options is

to notify the developers that the desired node can only execute a subset of their FFV chain. This technique leads to the second option, that is to communicate that fact but also expose other nodes that can run it integrally at the cost of a higher latency. The third option arises from the definition of the functions themselves. If these functions are defined generically, nodes only execute the subset of the function that the node's capabilities can handle. Another option consists of developers adapting a preexisting function to perform the desired task in the available nodes. For instance, imagine an FFV computing the average of sensors measurements and comparing it with historical values within a database. If a Fog node does not have the resources to perform a database query, developers can take that FFV and perform only the average computation of all the sensors measurements at the closest Fog node.

In consequence, FFVs enable the interoperability between the different players of an IoT system while hiding all their complexity to the users. FFVs create a dynamic architecture that can reuse components and thus enabling new business models for Fog-based systems. The time to market is then reduced and obstacles for new deployments are eliminated since just the non-existent functions need to be implemented.

Combined innovations: Resulting from the three innovations, new services do not need to develop the entire software stack. Now, the requirements are the development of the FFVs in case they are not already implemented, request an instance with the desired resources, and start executing the application. The result is the democratization of the IoT applications by reducing the entry barriers. Small companies can offer their services to end users without being constrained by operational expenses, similarly as what happens with mobile devices' applications.

3.4 Case Studies Analysis

To demonstrate the utility of the innovations in dealing with IoT applications, we present a two scenario case study. It reflects different situations encountered when deploying new services over a generic Fog-based infrastructure.

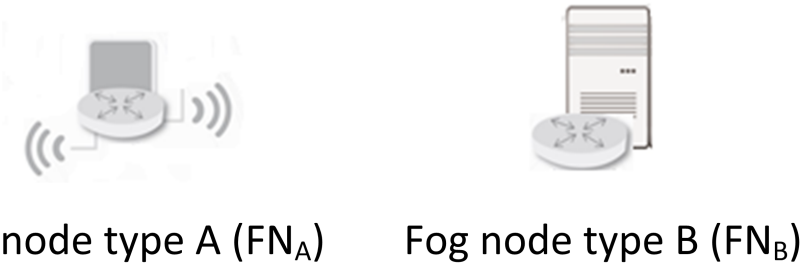

Figure 3.3: Two illustrative categories of Fog nodes named FN_A and FN_B.

3.4.1 Fundamentals

Before explaining the details of each scenario, it is necessary to explain the illustrative underlying infrastructure we use for this study. There are two types of Fog nodes as depicted in Figure 3.3. FN_A consists of wireless access points with some computational power. Its main advantage is an excellent connectivity with "things" while its drawback is its reduced set of computational capabilities. In contrast, FN_B consists of a server with some network cards. This node does not have wireless communications but has available an excellent computational capacity. Although Fog nodes such as FN_B can have a rich set of functionalities (i.e. CPU power, connectivity), they do not operate as a distributed datacenter. Nodes can form constellations to increment their available resources but there is no awareness of belonging to a global datacenter.

These nodes are deployed hierarchically on a smart city environment, supporting different working applications. Among these services there are connected vehicles and automated homes. Each application has deployed sensors to ensure their proper operation, integrating them into the infrastructure. For example, the cars have a set of sensors that monitors the pollution, sense their near environment (i.e. other cars or pedestrians), and monitors the engine behavior.

Additionally, there is a generic FFV implemented. Operating through a standard interface, this function reads as inputs the sensors' measurements and provides their average in real time as output. We assume all sensors provide 64-bit floating point values with the same granularity.

3.4.2 Scenario 1: Using available resources

The idea is to deploy a smart grid use-case to enable a more efficient use of renewable energy sources [23]. After analyzing the current infrastructure, this new service can be provided using pre-existing basic blocks. More concretely, automated homes can provide the power consumption on all appliances in real-time, plus control over certain devices including air conditioner, refrigerator, and electric car among others.

Using the predefined FFV, the application obtains the average power consumption per home. Later, this value can be computed by regions such as districts in the city. Based on these requirements, a vertical constellation – with Fog nodes on different hierarchical layers – is created as shown in Figure 3.4. Different FN_As are used to compute per-home consumptions, while the FN_B calculate the region's average due to their larger visibility (FN_B scope englobes a set of FN_As). Thanks to the modified orchestrator, only the necessary data is sent to upper layers although all the computation could take place at the node with more visibility. In consequence, the constellation hides the infrastructure's complexity and the application only sees its requirements fulfilled through its instance. In this case, Fog is chosen over the Cloud because of its operational capabilities more than to Cloud latency problems [79].

The other important part is to monitor the energy production. Each renewable source can provide the power generated in real time, processed with Fog nodes in a similar way the consumption is. These sources can be distributed geographically, solar panels can be in the city but windmills are outside. In consequence, the match between consumption and generation takes place in a distributed fashion at nodes where both values are visible. For example, in Figure 3.4 FN_{B2} monitors more energy produced than consumed in its coverage area. This remainder can be distributed to another region for consumption without polling each generator individually.

In addition, they want to activate the home appliances avoiding peak hours to benefit from lower energy prices. To reach this goal requires to define an FFV to provide analytics, mainly the energy consumption and generation per hour. Once this function is implemented, the nodes with control over energy sources

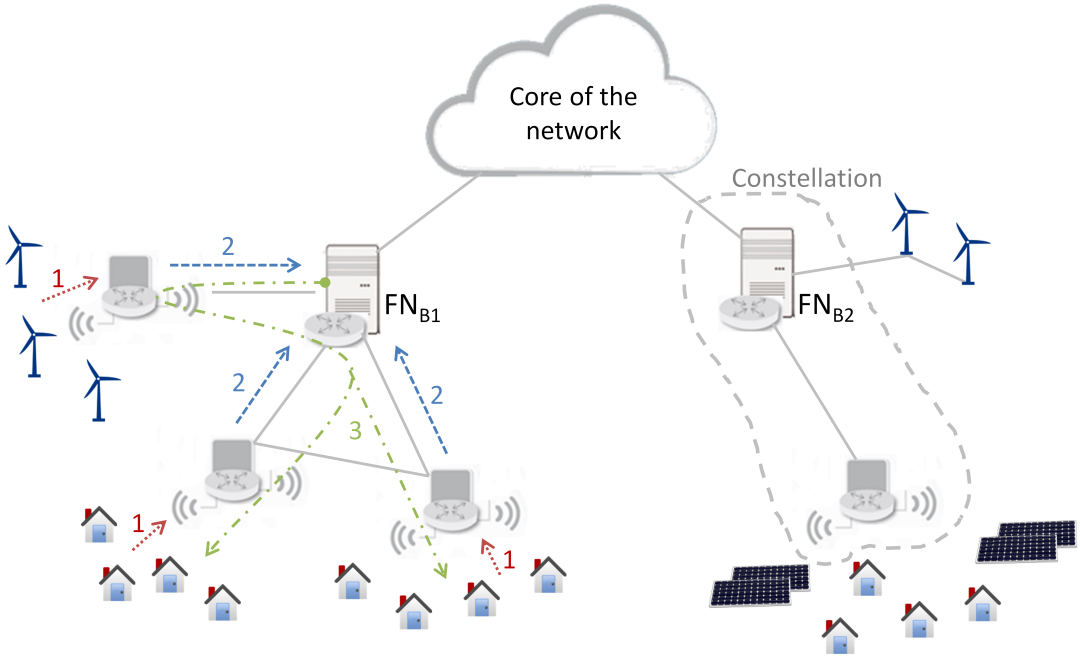

Figure 3.4: Example of a Fog architecture using the three innovations for a smart grid application. The system forms vertical constellations and the orchestrator involves two Fog layers.

can trigger the nodes controlling houses to activate appliances, as shown in the left branch of Figure 3.4.

3.4.3 Scenario 2: Adding resources to the infrastructure

Another application aims to provide a real-time contamination map including pollution and noise levels. In this case, connected vehicles proportion the pollution measurements with a key differentiation, mobility. While the cars move around the city, FN_A and FN_B nodes remain fixed. This fact provokes that a node cannot establish static associations with a certain set of sensors. Instead, each node covers an area and sensors migrate from one node to another, as reflected in Figure 3.5 when the car changes its Fog node from position P_1 to position P_2.

To gather the missing information, the application needs to deploy noise sensors across the city using streetlights. Afterwards, this new equipment can be integrated into the generic Fog-based infrastructure. In this way, the infrastruc-

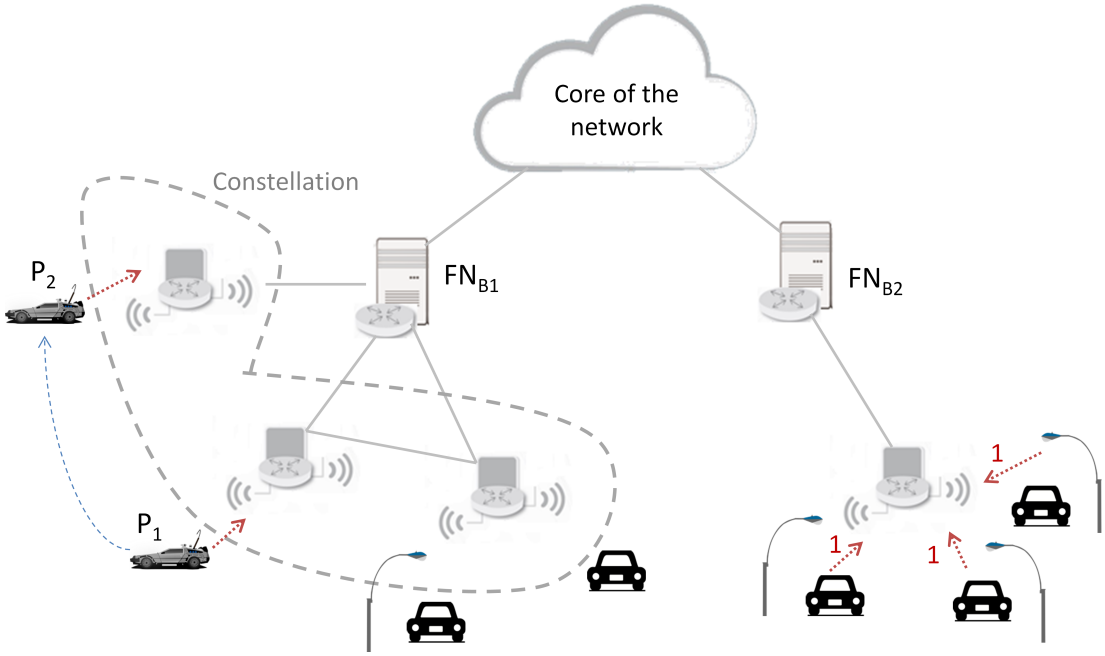

Figure 3.5: Fog infrastructure hosting a different application, a contamination map that includes pollution and noise levels. In this case, the system uses horizontal constellations and the orchestrator only exploits the first Fog layer.

ture capabilities improve with each deployment, becoming more attractive to new services. Now, the available system can provide all the contamination information to populate the map. Since this information can be displayed upon each sensor location, there is no need for higher aggregation levels. In consequence, the orchestrator function is simpler than in Scenario 1.

Based on the aforementioned requirements, horizontal constellations can handle this application as shown in Figure 3.5. This service could stop here but the predefined FFV can serve another purpose, to improve the sensor's accuracy. Collecting the values from nearby sensors allows to use their average value as the real measure. The precision increases as a function of the number of sensors and their type (i.e. not all the sensors will be equal or have the same error). The dynamism affecting nearby sensors seems challenging, but FFVs provide a flexible framework to deal with it. With a function implementing a discovery process, each node can determine the sensors under its influence. Then, and based on that information, the node applies the average function.

3.5 Summary

In this chapter we remarked the true potential of Fog by becoming a generic platform to support multiple applications simultaneously. To reach this objective, it is necessary to break the barriers that prevent its growth. The main obstacle is the amount of software required to integrate the Fog nodes into a current infrastructure.

To overcome this problem, we propose to enhance Fog with three innovations. First, to modify the orchestration policy allowing to execute more workloads on the Fog layers. Second, to create constellations of nodes to aggregate their capabilities, increasing the computational resources at the lower levels of the hierarchy. Third, the definition of the Fog Function Virtualization concept that provides great flexibility and adaptability. Now, the infrastructure's capabilities such as the "things" can be offered as functions and thus re-used for other applications. This solution brings Fog's democratization, enabling new applications to be deployed through the implementation of FFVs. Lastly, two scenarios were presented in a case study to show different implementations over a Fog-based platform.

Chapter 4

Hierarchical Emergent Behaviors (HEB)

4.1 Introduction

Internet of Things (IoT) includes a pervasive presence of sensors, actuators, and other devices that are deployed across large areas and connected via protocols (e.g. Bluetooth, WiFi, LoRA, 5G) that cooperate to meet common objectives [11]. The dominant characteristic of IoT is the physical interaction of the "things" with their environments, which enables novel applications and sets new architectural demands. Many of these applications are widely distributed, some have stringent real time requirements, and in all cases it is necessary to maintain trustworthy communication and adaptability to dynamic environments. IoT has the potential to significantly transform and improve city services, transportation, agriculture, health-care, energy production and distribution, and water conservation, among many other vital aspects of human life.

Conventional IoT deployments based on the simplistic approach of directly connecting "things" to the Cloud end up creating "silos" which limits the interoperability between applications. This approach complicates their orchestration and management, increases deployment costs, and it definitely does not support the scalability required to support autonomous vehicles, smart cities, transportation, and other relevant applications and services of interest. These applications are in fact Ultra Large Scale Systems (ULSS) [50], akin to the Systems of Systems

(SoS) that, based on Maier's definition [41], are an assemblage of different components where each is both operationally and managerially independent. There are compelling reasons to decentralize ULSS. They include manageability, which complexity grows with the scale of the system; and the ability to contain failures.

To tackle the scale and complexity of these ULSS we propose Hierarchical Emergent Behaviors (HEB), a paradigm that fuses the fields of emergent behaviors and hierarchical organization. Emergent behavior, previously studied in numerous fields, including social behavior, biology, and ethology [36], has lately gained traction in robotics and autonomous vehicles [37]. The key idea is to induce self-organizing behaviors akin to the swarm formation, thus bypassing arduous, centralized, and potentially brittle control mechanisms.

While current IoT systems rely upon extensive coding to achieve an explicitly-programmed behavior, we propose imposing only minimal and lightweight local rules to regulate "things" interactions in order to achieve objectives through emergent behavior. Unlike swarms, the ULSS of interest operate at different levels in terms of space and time. This observation suggests the consideration of a hierarchical organization, in which level (N+1) abstracts the behavior of level (N) while widening its spatial-temporal scope.

This fusion of emergent behavior and hierarchical organization concepts enables inducing desired behaviors without the need to envisage nor explicitly program for the vast number of potential scenarios. A well-designed HEB promises to be more flexible and adaptable to unanticipated conditions than a traditionally hard coded system.

The goal of this Chapter is to design and to develop an architecture that tackles the complexity and the scale of IoT ULSS. Once this is accomplished, different IoT systems can collaborate to become a real ULSS where each IoT system becomes a component of a larger entity. To this end, we combine the concepts of emergent behaviors and hierarchical decomposition to impose only lightweight rules over "things" rather than explicitly program the vast number of scenarios in advance. Thanks to HEB, devices can adapt to the continuously changing situations they face in complex environments by exploiting the information they have available and giving them more decision capacity. In consequence, we avoid arduous centralization mechanisms that precludes the system's scalability. After

consolidating a theory, we perform simulations to support our architectural assumptions revealing a great flexibility and adaptivity of the resultant HEB-based system.

In summary, the main contributions of this chapter are:

- We remark the orchestration, management, and scalability issues posed by IoT ULSS
- We propose the concept of HEB to tackle the aforementioned problems fusing the concepts of emergent behaviors and hierarchical decomposition
- We show how Fog Computing can enhance the induced behaviors thanks to its hierarchical structure that gives a larger vision of the system and the information available in the environment.
- We advance HEB to move from a concept to a more solid theory with a clear applicability to existing IoT deployments
- We simulate an Autonomous Vehicle scenario to validate the concept, the use of Fog Computing, and the addition of new rules, highlighting its potential when applied to IoT ULSS

In Section 4.2 we describe the fundamentals of HEB and its major advantages. Section 4.3 details what architectural modifications are necessary to implement it in modern IoT deployments and the majors challenges architects face. In Section 4.4 we perform a first evaluation of this concepts to prove its effectiveness and to highlight its potential. Section 4.5 explains how Fog Computing can contribute to enhance the induced behaviors by improving the availability of contextual information thanks to its larger vision of the system. In addition, we evaluate its impact through simulations. In Section 4.6 we advance the HEB theory consolidating the concepts and given the necessary steps to move from a concept to a more solid theory. We also perform simulations to evaluate these steps and to develop new rules that induce more complex behaviors. Finally, Section 4.7 concludes and summarizes our analysis.

4.2 Emergence in IoT

Emergent behavior may be defined as the collection of actions and patterns that result from local interactions between elements and their environment which have not been explicitly-programmed [42]. The local interactions themselves are driven by a set of engagement rules resulting in emergent behaviors and self-organization [14]. For instance, in his seminal paper Reynolds [54] defines flocking as the behavior that emerges when birds individually apply three local rules. The first rule seeks matching the speed of neighboring birds (alignment), the second rule prevents collisions with other birds or external objects by prescribing a minimum "bubble" around each bird (separation), and the third rule imposes a maximum distance between neighbors (cohesion). While none of these rules explicitly defines a collective behavior, a flock nevertheless emerges as a result of each individual bird flying according to its three given rules.

Collective behaviors such as flocks stem from the application of judiciously chosen local rules that are generic and independent of a specific time or location. The behaviors that emerge are a function of applying these rules within a particular environment (space, time, and contextual surroundings) which, when taken together, can be viewed as environmental constraints. Examples of such constraints are the quantity, density, and velocity of surrounding birds as well as the environmental obstacles such as walls. Applying these rules, however, does not necessarily lead to completely unpredictable behaviors. For instance, by knowing that the birds adhere to a collision avoidance rule, one can correctly predict that no two birds will crash even if they appear to be on a collision course with one another.

Transposing the flocking concepts to IoT results in viewing the "things" as the birds which follow given rules. As with the case of the birds, it is necessary to identify the constraints and choose appropriate rules that will affect the emergent behaviors for desired IoT-based services. For instance, Varaiya [72] laid the foundations to create groups of autonomous cars, called platoons. In his work a leader is designated in each platoon to manage group actions such as turns. These platoons can result out of an explicitly-programmed behavior (i.e. Varaiya) or emerge from a set of rules applied to each vehicle as the basic element taking also

into account environmental constraints. The application of the three flocking rules to a set of autonomous cars, for instance, properly illustrates the concepts under discussion since the flock becomes a platoon of vehicles without the need of a leader.

Determining the constraints placed upon the "things" is a non-trivial feat that determines the balance between emergent and explicitly-programmed behaviors. Three factors compose the constraints imposed upon a "thing":

a) Capabilities: These are the functional attributes of an element. For example, an autonomous car may move in two dimensions, vary its speed, and measure pollution. The emergent behavior cannot extend beyond the limits imposed by the capabilities of the elements.

b) Rules: The rules govern the interaction of the elements with the environment and among themselves. What limits emergent behaviors is the strictness of the rules rather than their quantity. There are two types of hyper-parameters associated with the rules:

- Numerical parameters with a physical meaning (i.e., the separation distance to prevent collisions).
- "Weights" assigned to the rules in order to establish their precedence.

c) Environment: The numerous environmental factors at play can heavily influence how behaviors emerge in different settings. For instance, identical elements following similar rules may behave markedly different in a city compared to a jungle.

4.2.1 Decomposability and Hierarchical Emergent Behavior

The basic functionalities of a platoon are implicitly embedded in the local rules. The functionalities required of an ULSS, however, are more complex than that of a flock. Trying to define from scratch the rules required to enable the emergence of those behaviors would be a daunting task. Here, the pioneering work of Simon [65] on decomposability of large scale systems defines a hierarchy among subsystems that enables grappling with the complexity of a system effectively

via an incremental approach. While the subsystems in Simon's work had explicit behaviors, we seek to combine this concept of decomposability with emergent behaviors in order to tackle the complexities of ULSS.

To this end, we propose extending the original emergent behavior approach by applying a hierarchical structure and tiered rules to the behaviors resulting from self-organization. Regarding IoT, Taft [67] proposes a hierarchical organization of smart grid. The focus is on the data aggregation at the different levels, with no concept of emergent behavior.

Our approach based on Hierarchical Emergent Behaviors (HEB) is depicted in Figure 4.1. The individual physical "things" are at the bottom of the hierarchy. A behavior emerges (e.g., the formation of a platoon) as a result of the application of the set of level 1 rules. The innovation behind the HEB concept lies in the application of a new set of rules to the emergent behavior resulting from the previous level. From the perspective of level 2, each platoon could be a level 2 "thing", which must follow the local level rules. For instance, level 2 local rules may establish certain minimum distance between platoons, and limit the number of vehicles within a platoon, from which the corresponding behavior emerges. While level 2 loses granularity due to aggregation, its scope in space and time is more expansive. The scheme is recursive, providing coarser granularity but wider scope as the hierarchy is climbed.

The HEB approach exploits the locality of interactions and perceptions since each hierarchical level provides a different vision of the elements. In the case of autonomous vehicles, for level 1 vehicles "locality" relates to the vehicles made "visible" through their on-line sensors. This "local" view does not include information about the way platoons are moving along the highway. The 2^{nd} level, rather than detailed information about the interactions within each platoon, keeps track of the flow of the platoons as single elements. This abstraction allows implementing different regimes of operation exploiting the "locality". For instance, during the normal regime the goal is to keep certain metrics (i.e. distance between them) at the desired level by tweaking the parameters that regulate the 1^{st} level behavior. The anomalous regime kicks in when, due to its long vision, the 2^{nd} level detects the onset of a congestion that requires rerouting.

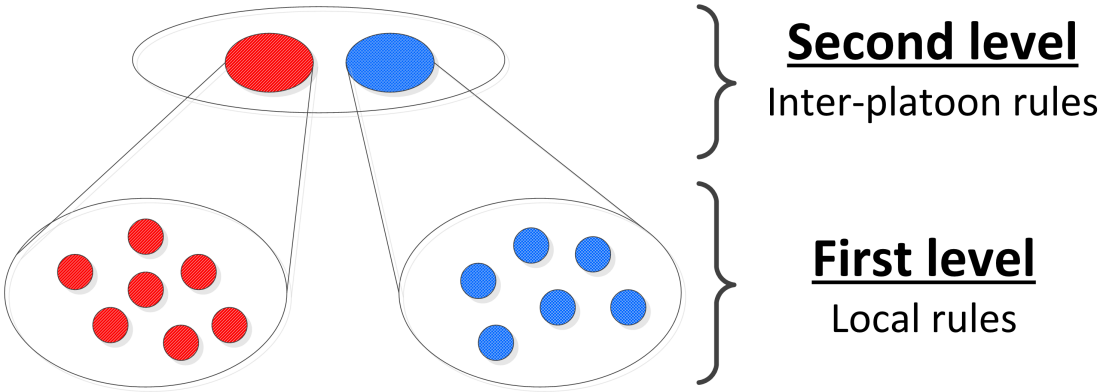

Figure 4.1: Representation of the HEB concept with different hierarchy levels. The first level rules applied to level 1 elements (e.g. vehicles) induce a platoon behavior. The second level applies inter-level rules over the previous level behaviors (e.g. platoons) to enable more complex functionalities.

4.2.2 Advantages of emergent behavior for IoT

The manner in which emergent behaviors cope with complexity and scalability issues is their main advantage. The functionalities as a system arise from local interactions and the element's flexibility instead of explicitly programming the uncountable number of possible situations.

This brittleness of the explicitly-programmed approach manifests itself even with systems composed of a few elements. The work by Saska et al. [63] used a specifically-programmed hawk-eyed supervising element that controlled and corrected the position of another set of autonomous vehicles. This central orchestrator is aware of the entire system and is responsible of maintaining the formation when faced with obstacles or failures. Consequently, it becomes the main bottleneck although the system is only composed of tens of elements. In comparison, emergent behavior eliminates the need of a central orchestrator that would have to deal with a very large number of "things". Moreover, the system's complexity is greatly reduced since decisions are taken in a distributed fashion, leveraging the intelligence of each "thing". Platoons emerge induced by the rules imposed over vehicles although each car is not aware of the size of the platoon nor are programmed to form groups.

Scalable IoT systems also need to adapt to the changes that occur in the dynamic environments where the "things" are deployed. Due to the large number of variables and situations, designing an explicit programmed system that takes into account all the scenarios in advance is a formidable task. With our HEB IoT approach and if the proper set of rules is defined, the "things" are able to dynamically adapt to the environment without the need to explicitly program them.

Lastly, applying HEB allows each higher level of the hierarchy to abstract away the complexities of the lower levels. This is the result of aggregating capabilities and data from individual "things", leading to less complex software development. For example, a level 2 "thing" (e.g. platoon) is composed of numerous level 1 "things" (e.g. vehicles). This process allows for a single query to be issued to a whole group instead of to each individual physical "thing".

4.3 Implementing Emergent Behaviors fo an ULSS IoT System

Current IoT architectures follow the structure depicted in 4.2. "Things" deployed for a single purpose are at the bottom of the architectural stack. These elements are sensors that measure their environment, and actuators that respond to commands. The connectivity layer manages the communication with the higher layers. Aggregation, curation, and in many situations processing (including deciding what must be handled locally, and what must be sent to the Cloud) are the main functionalities of the Data Layer. The application layer at the top defines and manages the tasks that compose the services.

4.3.1 Emergent Architectures

Implementing HEB on an IoT system requires modifications to this traditional application stack, impacting "things" and their communications.

In some IoT deployments "things" are passive sensors, and in many others they also include actuators. The actuation role gets heightened in HEB. In addition to their common operation regime, "things" will now take decisions based

on their interactions with other "things" and the environment. For instance, the decision to join other vehicles in a platoon can be taken by the cars themselves without involving a higher orchestration layer. Hence, by 'active' we understand any device that participates in the generation of emergent behaviors or alters them (i.e. applying rules, modifying the environment, etc.). Distributing intelligence to the "things" exploits their locality while potentially reducing the managing complexity of an ULSS.

To sustain this new architectural feature, the communication capabilities require major changes. In a traditional design, architects must anticipate every possible scenario, and the communication patterns they entail. HEB's approach is radically different, in that the scenario space is explored using realistic simulations, but no claim is made that every single emergent behavior is covered. Rather, the effort is focused on ensuring the correctness of the local rules at the different levels of the hierarchy. Then, what matters in HEB is the interoperability between "things", achieved through standardized APIs and interfaces. Once "things" can communicate with each other and with their environment, new behaviors emerge by the application of the local rules. For instance, vehicles can be of different brands, and use a diverse array of sensors. Picture a vehicle communicating with other cars to join their platoon. If cars cannot communicate, the resultant behaviors would be very limited since not all the vehicles can become part of them. Instead, if the interoperability between cars is guaranteed, any vehicle could become part of a platoon. In addition, this vehicle could also communicate with sensors placed on streetlights to obtain contextual data that could improve the platoon's efficiency. Enabling these communications becomes critical to ensure the success of emergent IoT architectures.

A side effect of this communication can lead to an important advantage, the aggregation of data from heterogeneous sensors. Grouping sensors could lead to new functionalities without deploying new hardware. Imagine a platoon where each vehicle has a different set of sensors. Thanks to the abstraction provided by HEB, the platoon's capabilities will be the sum of each vehicle features. For instance, one vehicle measures the temperature and the following measures the pollution. In this case, the platoon can provide both measurements, hiding the

fact that each vehicle only provides a single measurement. If the system aggregates sensors of the same type then the measurement's accuracy can improve by providing their average. In this case, a filter based on historical data and accuracy determines which sensor value is more likely to be correct.

The aforementioned changes translate into the addition of a new layer to the traditional IoT stack, the Rules layer, as shown in Figure 4.2. It is responsible for the rules and the hyper-parameters controlling the local interactions while maintaining the communication among "things". A different sub-layer handles each function. "Hierarchical Rules" sub-layer deals with the rules in each of the HEB levels and how they are applied over the physical "things". "Things communication" sub-layer manages the type of "thing", spatial and temporal information, security policies, and hierarchical queries and responses. The connectivity layer provides a channel to communicate resultant emergent behaviors with higher layers, while "Things communication" sub-layer focuses on the communication among "things" themselves to induce behaviors. Once a first level behavior is obtained, applying HEB converts the entire application stack into a second level "thing". In this case a new set of rules can be applied to the second level "things" in order to obtain more complex emergent behaviors. These behaviors end up in applications with more functionalities while maintaining the dynamism of the previous level "things".

4.3.2 Challenges

HEB holds tremendous potential to design and orchestrate ULSS, but the promise requires overcoming new challenges. Particularly critical challenges include: (i) behavior shaping, (ii) reliability, (iii) intra- and inter-level communications, and (iv) security.

4.3.2.1 Behavior Shaping

An intrinsic characteristic of emergent behaviors is that architects induce new behaviors by implementing different sets of rules [42] as well as by varying the complex environments. After applying a set of rules the architect performs a

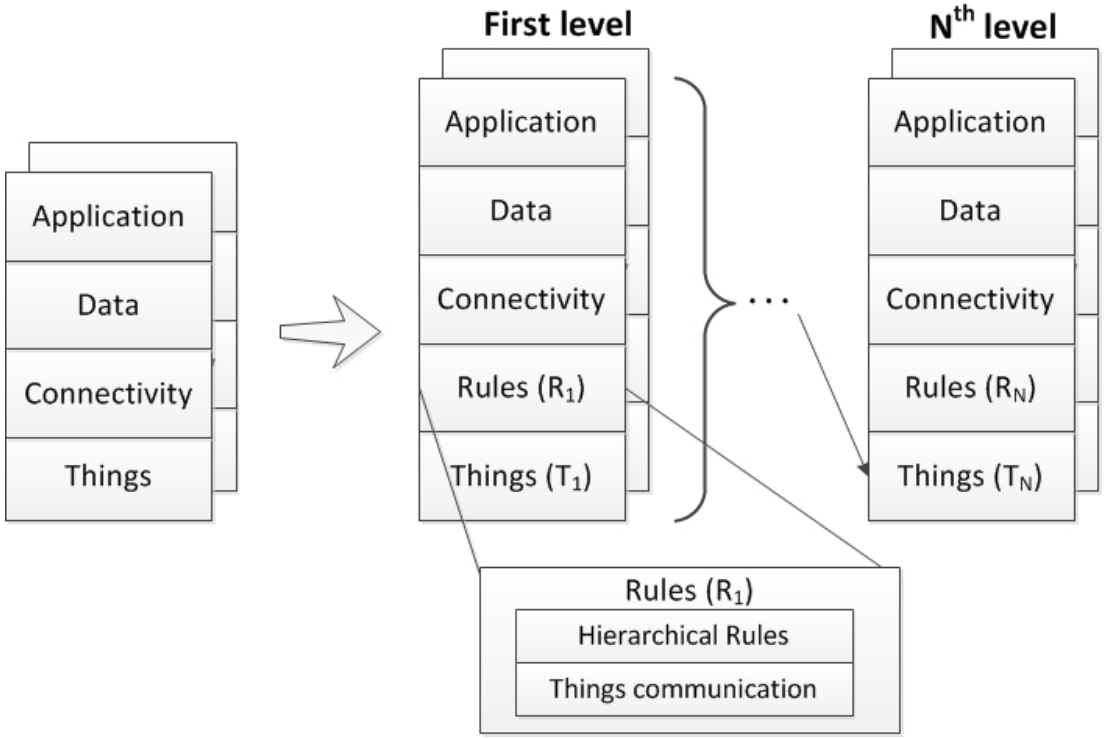

Figure 4.2: HEB requires the layer "Rules" between the classical stack layers "Things" and "Connectivity", at every level of the hierarchy. For instance, if N=2, emergent behaviors out of level 1 become the level 2 "things". Then, the level 2 rules complement the level 1 rules.

selection process to determine the useful behaviors. This decision becomes critical since it determines the application's functionalities. Non-useful behaviors are discarded and the rules adjusted correspondingly. Conversely, rules that are responsible for the useful behaviors can be enhanced and used more often.

Different optimizations can be applied to rules generating useful behaviors. One option consists of adjusting the rules' parameters to tune the behavior. For example, in a platoon we can adjust the collision avoidance distance so it covers a larger area. Another option is to add/remove rules to modify behaviors. The criteria to decide which behaviors are useful can be based on performance, formed groups, types of "things", and criticality among others. The result of this selection determines how effective the emergent IoT system will be.

4.3.2.2 Reliability

Provided that IoT is built upon millions of non-reliable low-cost sensors, we can expect a large number of failures aggravated by harsh environments. Fault propagation, similar to waves in water, should be stemmed. For example, Gerla et al. considered a platoon that suddenly stops for a crossing pedestrian [27]. Without proper boundaries, following platoons will also reduce their speed thus propagating the effect. A proper inter-platoon distance, controlled by 2^{nd} order rules, should avoid this problem.

A second problem arises when catastrophic failures occur due to bad behaviors. Instead of shutting down the entire system, there are two options. Either a functional backup is restored or a supervisor takes control of the system momentarily to manage the ULSS.

4.3.2.3 Intra- and Inter-level communication

Communication among "things" from different applications is a challenge by itself. Although identified years ago, it is not solved yet. HEB accentuates the urgency. For example, a platoon moves near another. To apply the second level rules (inter-platoon rules), it is necessary for both entities to communicate as level 2 "things". Open literature usually designates a leader to perform this task. However, centralization results in the loss of the locality and scalability advantages of HEB since a vehicle does not know the size of the platoon nor has membership awareness. Defining how entities from different levels communicate and how the rules interact will determine the performance of the behaviors.

4.3.2.4 Security

Emergent behaviors offer significant advantages in dealing with the overwhelming complexity of ULSS. However, they also extend the "attack surface" that can be exploited. An attacker that gains access could modify the rules, either directly or through modification of the hyper-parameters, for nefarious purposes. There is no magic bullet in security, but there are three major recommendations to follow: a) security is not add-on, incorporate it as an integral part of the design effort; b)

leverage HEB context awareness to detect intrusions and other forms of attack; c) make sure that the design has the ability to isolate infections.

4.4 Initial evaluation through simulations

4.4.1 Fundamentals

A group of autonomous vehicles, either aerial (i.e. drones) or terrestrial (i.e. cars), constitute the basic elements of this system. For us, a vehicle is a sensor platform that applies rule-inducing behaviors as a single object, although it could be decomposed into its own component "things". Cars measure ambient temperature and pollution while drones focus only on pollution.

Each vehicle implements the three original rules from Reynolds (R_1 Alignment, R_2 Separation, and R_3 Cohesion). In addition, they have a rule to reach a target destination point (R_4 Destination) and a level 2 rule to induce a platoon of platoons (R_5 Second). To highlight the complexity of each rule, R_5 performs the same operations as R_1, R_2, and R_3 combined but applied to level 2 "things". Moreover, each rule is weighted by a value that can be modified (or even deactivated) in real time to observe their impact on the induced behaviors.

With these set of rules, the vehicles are ready to circulate in a city. This is a dynamic environment with limitations to their mobility (streets and obstacles) and other working IoT systems that translate into a huge amount of possible interactions and situations.

4.4.2 Methodology

We chose the Processing simulator [6] to perform our analysis of flocks. We modified the base flock example to add the new rules and constraints. As a consequence, each vehicle determines its trajectory based on the five implemented rules guided by its local interactions. Each type of vehicle is represented with a different color that also indicates its mobility patters (i.e. drones can fly over obstacles) and R_2 is modified so cars avoid their surrounding obstacles as well. To represent streets and other orographic patterns, we use obstacles shown as black dots in the canvas.

With these modifications, Processing offers a rich framework to simulate and visually observe vehicles, their environment, and their interactions. We also implement a mechanism to detect incorrect behaviors due to the violation of the rules (i.e. a car passing through an obstacle), complemented by the visual validation using the simulation canvas. The overall performance of the system is evaluated using both the visual observation and the actual alarms.

4.4.3 Emergent Autonomous Vehicles

The simulation begins when we place a set of autonomous cars on a side of a straight street and specify the destination (through R_4). After a certain time, a platoon of cars is formed induced by R_1, R_2, R_3. For now, R_5 is deactivated (weight = 0) to focus on the level 1 behaviors.

Circulating as a platoon, the vehicles face a part of the street full of obstacles (i.e. non-emergent vehicles). At this moment the platoon behavior dissolves because each car focuses on avoiding obstacles (R_2 prevails over the rest for safety), as reflected in Figure 4.3. This illustrates the effect of a changing environment on behaviors. Here cars were the moving element although it could have been reversed. Consider, for example, an IoT system of deployed sensors on lampposts to monitor certain traffic patterns. Therefore, "things" are static but they react to mobile elements in the environment (i.e. cars).

Once cars overcome the obstacles, their self-organization again results in the formation of a platoon. Next, we modify the separation distance in R_3 to observe the impact the rule's parameters have on the behaviors. Increasing this parameter results in the dispersion of the platoon, which may be desirable in search and rescue applications, for example. On the other hand, inadequately tuning these parameters may result in no or little constructive emergent behaviors. This shows that changes in the rules' hyper-parameters (weights and numerical values) greatly affect behaviors. In consequence, it may be desirable to adjust them in real time to efficiently deal with the dynamism of the ULSS.

The platoon now faces an intersection with another platoon on the road to its left. Since R_5 is still deactivated, they interact as level 1 "things" and not as

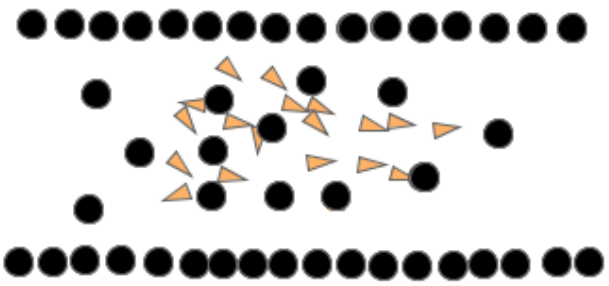

Figure 4.3: Emergent behavior shaped by the street and obstacles. The platoon does not emerge due to the environment.

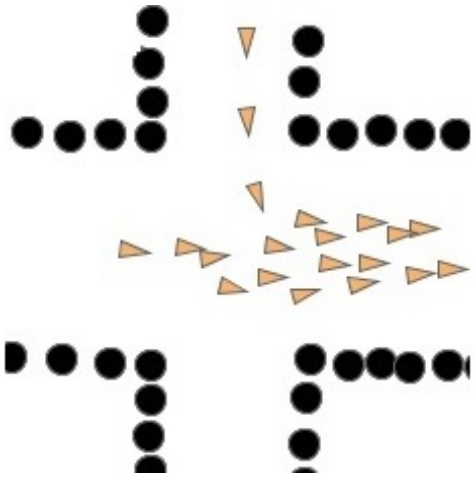

Figure 4.4: Two platoons approaching an intersection without an orchestrator (traffic lights). In this case they form a larger platoon.

platoons. R_4 determines whether the two platoons will form a larger grouping or each will continue their way, depending on their destination targets. In any case, the local interactions among cars prevent any collisions thanks to R_2. Since they have the same destination, they indeed form a larger platoon and continue advancing, as shown in Figure 4.4. Following these rules, there was no need for a traffic light system to act as a global orchestrator between vehicle traffic.

Finally, the platoon reaches the end of the street and faces an unbounded area (no street delimiter). This platoon encounters a flock of drones induced by R_1, R_2, R_3. Although level 2 rules can apply among platoons of the same "things", we used aerial autonomous vehicles (drones). If we now activate R_5, then when the flock and the platoon are close enough, they will form a second level group, inducing new emergent behaviors. The cars in the platoon now use the capabilities of the drones inside the flock to scout the optimum route to avoid traffic and obstacles. Conversely, the drones in the flock may use the street level sensor data to increase their pollution measurement accuracy. Their level 1 functionalities are still preserved even though now they share information as level 2 "things".

Figure 4.5 shows this situation. Cars (in red) and drones (in orange) can overlap since drones fly over them. The drones forming part of the level 2 group change their color to blue to indicate this new service. Not all the drones are part of the second level group due to the physical separation with the cars, which adds flexibility to HEBs. Once this distance becomes too large they split off, returning to level 1 behaviors only. Then, R_5 enabled the interoperability of different applications to provide new functionalities as a system.

4.5 Fog Computing in support of HEB

The stringent latency requirements associated with autonomous vehicles suggests distributed platforms rather than the Cloud for their management [62]. Fog Computing [15] has long recognized the value of extending the Cloud to the edge of the network, bringing networking, compute, and storage resources at different hierarchical levels to respond to the needs of applications and services. Fog addresses the infrastructure and orchestration issues regarding the computational resources [64] (i.e. processing, storage, communications) both at the edge and at different levels of the hierarchy.

Fog can support HEB providing contextual information with a larger scope than that of the on-board sensors. Thanks to the Fog hierarchy, this platform has

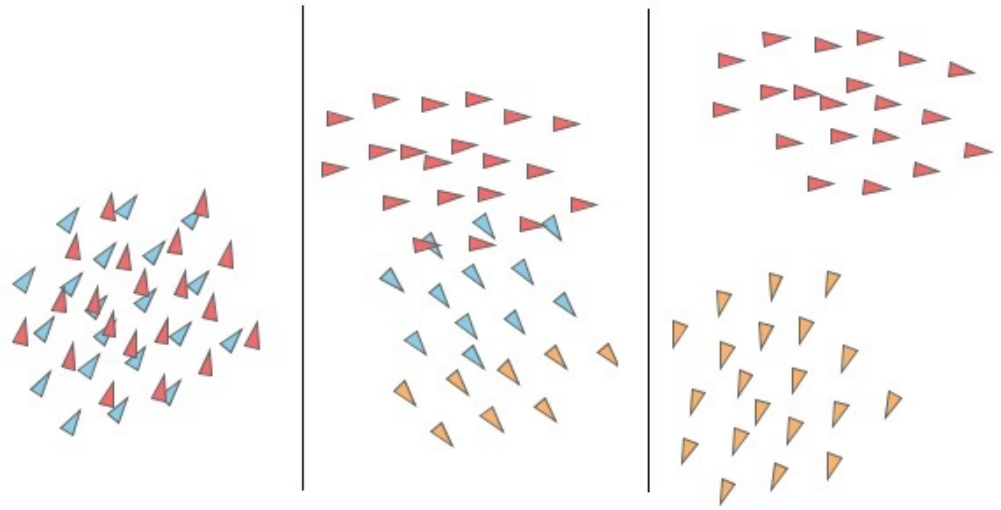

Figure 4.5: Sequence of interaction between a flock of drones and a platoon of cars based on second level rules.

a better vision of the vehicles' environment. When a relevant event is detected, Fog nodes can transmit that information to the cars to enhance their reaction to such event. For instance, a node could add a new rule to induce behaviors that are more resilient when facing an event.

Another advantage Fog brings is the visibility within the behaviors. Fog nodes placed along the roads can passively measure and observe the induced behaviors and report them to the users, enhancing the visibility of the system.

Previous Sections outlined an agenda to deal with ULSS, with emphasis on AVs. This Section advances the agenda in several significant ways: a) developing the concept of "emergent behavior primitives", and studying the maneuvers of vehicles exiting a platoon and anticipating to obstacles beyond sensors range; b) emphasizing the role of Fog Computing as support for HEB communications in general, and facilitating contextual awareness in particular.

4.5.1 HEB, the next phase

The application of the three original rules from Reynolds [54] to a set of autonomous vehicles results in the formation of a platoon [72] without explicitly

program that behavior. However, these rules do not specify the absolute velocity of the group. Platoon absolute velocity is defined as the absolute average velocity of all the vehicles forming the platoon. This velocity is a crucial metric in autonomous vehicles and highly depends on the context, including the quality of the road, weather conditions, vehicle density, maneuvers, and neighboring platoons among others.

The above considerations strongly suggest the need to define the policy not only in terms of local rules. At the end, a policy maps the information state of the system into an admissible set of decisions. For AV HEB, a policy at any given level of the hierarchy includes:

- Local rules pertaining to the hierarchical level.
- The set of hyper-parameters associated to those rules. This set not only includes parameters such as the rules' weights, separation distance, etc. but also velocity applied to each level (i.e. first level refers to average speed of the cars, at the second level is a vector of velocities for each platoon)
- Contextual information. The challenge is to capture in a succinct way the critical information. This requires analysis and careful experimentation. The issue is the required degree of granularity. Contextual awareness includes car density, weather conditions, road conditions, platoon regime, etc.

Architects can define a policy portfolio with well-known emergent behaviors to implement. Given that contextual information is captured in the policies, the selection process becomes a simple, even a trivial one. There are only a few admissible policies for a given informational scenario. Then, the first set of policies to define are the so called "emergent behaviors primitives".

4.5.1.1 HEB primitives

By primitives we understand basic operations required by vehicles within a platoon. Right now we focus on first level behaviors, but the same concept applies to any level within HEB. Vehicle maneuver without collision or handling autonomous

cars that want to take an exit in a highway constitute primary examples of a primitive [72]. Simple as they sound, this requires consideration of different aspects and interactions of HEB components:

- Communications between different entities: (i) vehicle to vehicle, (ii) vehicle to RSU, and (iii) distribution of functionalities within the platoon
- Vehicle announcement of its intent
- Non-intersecting exit trajectories whether one or multiple vehicles leave the platoon
- Emergent behaviors at play: current operating rules, their hyper-parameters, and new individual behavior (i.e. exiting the platoon) affecting the emergent behavior.

Taking a closer look to these aspects of the maneuver without collision, we observe the effect of the emergent behaviors through the separation rule between "things" and obstacles. In this case, a single rule provides us the primitive objective if the proper sensing capabilities are satisfied in each moving vehicle.

The exiting highway maneuver requires more considerations. Although each vehicle does not know the number of vehicles in the platoon nor has membership awareness, it may notify to its neighboring cars its exit. There is a fundamental reason for this notification: to avoid an undesired behavior with the entire platoon unconsciously following the exiting vehicle/s.

Communications constitute a key element to ensure a satisfactory maneuver. Efficient exiting strategies necessarily rely on contextual information. It is useful to distinguish between permanent information (coordinates of the exit, proximity to other exits, etc.) from real time information (state of the road, congestion level at the exit, weather conditions, speed of the platoon, vehicular density).

Fog Computing is of great help, from the compute and storage capabilities of the RSU at the edge, to the exchange of real-time information along the RSUs. Fog nodes then become the RSUs in the roads providing their capabilities to the vehicles while building applications on top of their contextual information (i.e.

smart guidance systems). Another alternative could be to use the same vehicles to detect and classify the lanes [47].

Last but not least, the use of non-intersecting trajectories in the 3-dimensional space are a must. In traditional solutions with explicitly programmed behaviors, a central orchestrator determines each vehicle's trajectory and makes sure no collision happens. Instead, HEB defines a rule to prevent collisions and gives freedom of choice to the vehicles to decide the best trajectory based on their contextual information. Figure 4.6 depicts these differences between the two methodologies. In opposition to preset trajectories, HEB approach creates local rules that lead to behaviors emerging in the form of trajectories.

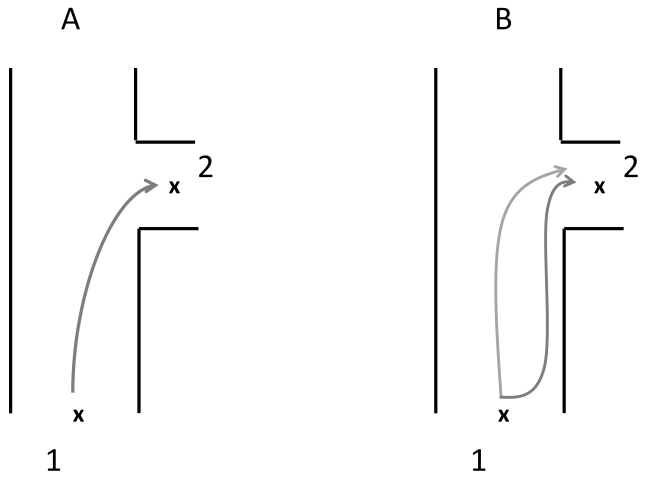

Figure 4.6: Left figure shows an specific trajectory that a vehicle must closely follow. Instead, the right figure shows the freedom HEB leverages to the vehicles to decide their trajectory. Vehicles are capable of taking best decisions since they have all the contextual information.

4.5.2 AV primitives case study evaluation

AVs constitute one of the clearest HEB applications. The mobility of the vehicles, the constantly changing situations (i.e. road conditions, weather, traffic conditions), and the number of cars provide a rich set of interactions from which behaviors emerge.

The universe of AV maneuvers is the composition of a vast set of "primitives". We approach the validation of the HEB architecture by creating and examining in detail a rich "library" of primitives. Each primitive is defined by a goal, and it is self-contained in that it has the ability to reach said goal. We envision the creation of complex scenarios by concatenating primitives. More precisely, we will consider richer goals that are the composition of simpler goals, and achieve them by concatenating primitives. This Section, which is the first step in this direction, focuses on two primitives that rely on Fog nodes deployed as RSUs.

While a primitive is defined by a goal, its full characterization necessitates the specification of the rules that facilitate the achievement of the goal.

The two primitives under evaluation are the exiting maneuver in a highway and anticipating and reacting to obstacles beyond the sensors range. There are many ways of implementing a primitive. In the following Sections we perform a preliminary analysis and present tentative solutions that satisfy the objectives of these two primitives.

4.5.2.1 Exiting maneuver

In this primitive one or more vehicles in a platoon decide to leave the platoon and exit the highway. We consider the primitive is accomplished satisfactorily if the vehicles exit without collisions or hazardous maneuvers and the rest of the platoon continues the journey unperturbed [34]. Exiting brings forth the interplay between emergent behavior, classical trajectory design methodologies, and inter/intra layer communications:

- Communications: they were explained before in detail, mainly focused on V2V and V2I [4] enhanced with each vehicle's sensing capabilities.
- Emergent behaviors: Platoon behavior (induced through the three original Reynold's rules) and moving objects start roughly at the same speed along the road. As the exiting maneuver proceeds, the velocity vector of the moving object changes in direction and magnitude, but not in a brusque way.

- Trajectory design: Vehicles in the platoon as well as exiting vehicles are interacting but autonomous decision makers, in that they sense the environment and react accordingly. The strategy of fixing exiting trajectories and relying on the collision avoidance ability of vehicles in the platoon seems sound and straightforward. This strategy regards only the platoon vehicles as interacting autonomous decision makers, as depicted in Figure 4.6.

Among the possible implementations of the exiting primitive we analyze three possibilities of different complexity and observe their impact on the behaviors of interest. The first implementation starts with vehicles notifying their intent to leave the platoon. Since there is no central control of the platoon or membership awareness each vehicle needs to handle the notification to its surrounding vehicles. An intuitive way of notifying its intention is to change the vehicle's role within the platoon. Instead of being perceived as a vehicle, and therefore subjected to the three platoon rules (R_1, R_2, and R_3), perception changes to that of a moving obstacle. In this case, the rest of the vehicles within the platoon avoid it by simply following the non-collision rule (R_2). This technique results in exiting vehicles creating a virtual path within the platoon till they make their exit. The vehicle's new role allows it to leave the platoon and take the desired exit without compromising the platoon behavior for the remaining vehicles.

The challenge is to effect that change of role (from a vehicle in the platoon to moving object) without affecting the local rules or resorting to a central orchestrator. Visualize the scenario in which a platoon of vehicles moves along a highway as depicted in Figure 4.7. A RSU notifies the platoon of the existence of an exit ahead. The RSU is actually a Fog node, part of a full Fog hierarchy deployed along the highway. The Fog node keeps contextual information, including obstacles in the road ahead, congestion levels, weather conditions in the area, etc. as part of the rich information exchanged with other Fog nodes, both at the same and higher hierarchical levels. Hence, the Fog can extends a vehicle "vision" beyond the capabilities of the on-board sensors.

A vehicle decides to take the forthcoming exit, and broadcasts its neighbors the change of its role, from a peer in the platoon, to mobile obstacle. It does so through the V2V communication channel (e.g., DSRC). From that moment

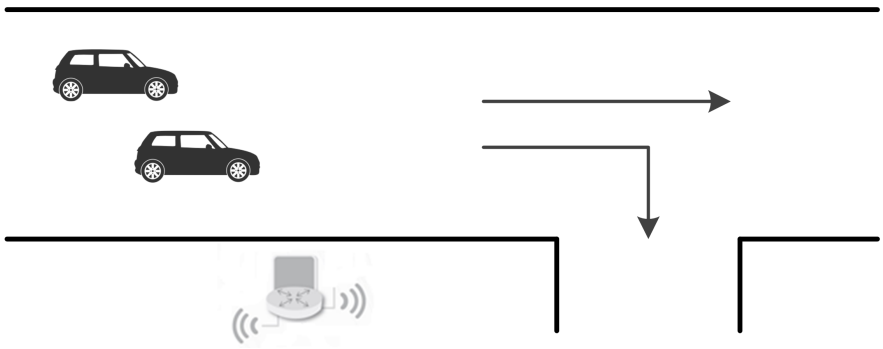

Figure 4.7: Scenario to evaluate the exiting maneuver. It consists of a highway with an exit. The autonomous vehicles can either exit or continue in the highway based on their final destination. The RSU deployed as a Fog node assists with the contextual information.

on, that vehicle is perceived as an obstacle by any vehicle happens to be in its neighborhood. As the exiting vehicle maneuvers, its neighborhood changes, but as the notification of its role keeps active, the new neighbors keep away from it. Hence, the exiting vehicle carves a wormhole through the platoon that leads to the exit.

The same methodology applies when more than one vehicle wants to take the exit. In this case, each exiting vehicle acts individually and it is not coordinated with the other exiting entities. We exploit the power of the rules and their flexibility. Since each vehicle avoids obstacles (R_2), there will be no collision among vehicles whether they are part of the platoon or they are leaving. There is no need to implement costly orchestration mechanisms to anticipate all possible situations in micro detail, we just give basic rules and let the vehicles decide what is best for them. The result is a set of vehicles "leaving" the platoon and taking the exit, while the rest of the platoon moves along the highway to its destination.

The second implementation uses on a more direct approach based on the rules and their hyper-parameters. This solution does not require extra communications (i.e. notifications) to achieve the primitive's objective. When the RSU announces the exit that one or more vehicles want to take, the exiting vehicles modify their destination target in R_4 and simultaneously modify the weight associated to that rule. Recall that weights define priorities among the rules that determine the

local behavior. The separation rule (R_2) still keeps the highest priority to ensure no collision happens but the destination rule dominates (R_4) over the rest (R_1 and R_3).

While this approach also produces the desired result (vehicles exiting the highway without collisions), we observe differences in the behaviors, which may affect the time it takes to exit the highway. Giving priority to the destination rule over the traditional platoon rules ensures that the vehicles take the exit instead of continuing as part of the platoon. Similar groupings based on destination were analyzed by Hall et al. [30]. What changes with respect to the previous case is how the local rules apply. While the previous technique is based on vehicle to object interactions, the second one relies on rules between vehicles.

We experimented with a third approach, which is in fact a particular case of the previous one: exiting vehicles modify their destination targets, but they do not alter the weight of the rule. The fact that this approach meets the goals of the exiting primitive highlights the surprising expressiveness of the local rules. Adding a target destination rule (R_4) we can induce many useful behaviors. Besides the obvious behavior of reaching a destination, vehicles with different targets can form a common platoon and later one split to reach both destinations.

Figure 4.8 depicts a temporal representation of the exiting maneuver implementing the third technique (the results from the previous two are the same except the aforementioned differences). The left figure shows the platoon at the beginning of the highway just before crossing the RSU that communicates the forthcoming exit. The center figure shows some vehicles "leaving" the platoon. Finally, the right figure shows the small platoon of exiting vehicles as well as the platoon of remaining vehicles in the highway. Simulation details not in the figure show exiting cars moving to the edge of the platoon, positioning themselves for a smooth exit, without vehicles crossing their paths. This behavior, not explicitly programmed, emerged naturally from the local rules. It is actually the result of some rules dominating others (in this case R_4).

The policy emerging with the third technique has considerable degrees of freedom. Consider for instance the rare case in which an exiting vehicle finds the exit suddenly blocked by vehicles ahead. The vehicle cannot force its way out because it is not acting as a mobile obstacle, but rather as a peer of the other

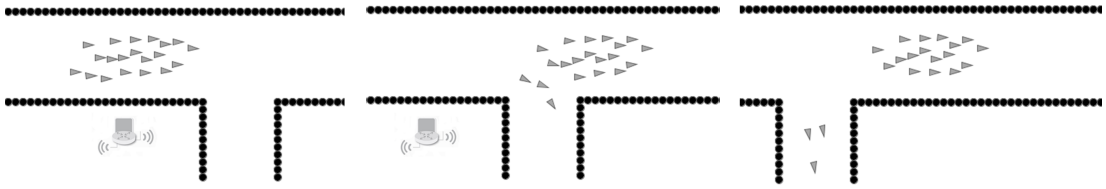

Figure 4.8: Sequential representation (from left to right) of a platoon executing the exiting primitive assisted by the RSU.

vehicles. The car can head back, and rejoin the platoon. Note that this would not be the case with the previous techniques, because they are more aggressive.

Another remarkable case happens if the platoon does not have a specified target destination at the end side of the highway. In this scenario, when the exiting vehicle executes its maneuver, the rest of the platoon can follow it. This behavior is not problematic though because each vehicle always have a target destination.

A simple set of four rules provides a wide range of useful behaviors that are very robust. In addition these lightweight rules demonstrate an incredible flexibility and simplicity. Fog nodes contribute to handle the contextual information. In this case it only transmits the target destination that will make the vehicles take the proper sideway. We have seen three different ways of implementing this primitive but there are more options that can be part of the policy portfolio (rules, hyper-parameters, contextual information) and can be reused for other applications, reducing the deployment time.

4.5.2.2 Anticipating and reacting to obstacles beyond the sensors range

In this primitive a vehicle or a set of them is circulating in a highway and there is an obstacle beyond the onboard sensors range. The main objective is to anticipate its detection and react accordingly to overcome it without compromising the safety of the driving. Overcoming obstacles brings forth the same areas as the exiting maneuver with the difference that this primitive directly targets the hyper-parameters within the emergent behavior rules.

This scenario is slightly different from the previous ones. We have a platoon moving along a highway as depicted in Figure 4.9. Along the road there are a set of RSUs that gather information about the road conditions, weather, and traffic among others. In summary, these Fog nodes manage contextual information related to the highway. These nodes are organized hierarchically to capture the information of a wider area. Figure 4.9 also depicts the virtual architecture they conform with two different levels. The first level is formed by the RSUs closer to the highway, the nodes that physically deal with the vehicles. On the second level there is a single RSU that aggregates the information from the previous level. This level 2 node has a wider scope but its granularity is coarser.

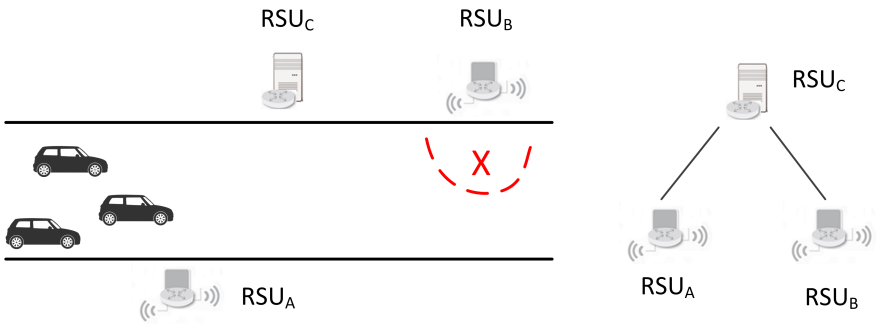

Figure 4.9: Scenario to anticipate and react to obstacles beyond the sensors range. It consists of a highway with a three RSUs organized hierarchically. The right side of the Figure details this architecture with two nodes at the bottom that directly communicate with the "things", and a higher node to aggregate all the data. The cross represents a temporal blockade in the road.

This hierarchical RSU configuration provides information to the vehicles that traverse the boundaries of locality. While vehicles only sense its closest surroundings, RSUs have a larger scope and can transmit that information to the vehicles. In this situation, vehicles can prepare for the upcoming obstacle or other events. The procedure is as follows: RSU_B senses the blockade and besides notifying it to the vehicles within its range, it sends this information to the higher layer node, RSU_C. Then, this node can transmit the information to the other lower level RSUs to take proper actions. In this scenario, RSU_C sends the notification to

RSU_A that is the node at the left. Now, RSU_A has the information on the road status ahead and can notify it to the nearby platoon.

To optimize the reaction of the platoon to the blockade, RSU_A acts upon the hyper-parameters modifying the separation distance between the vehicles and also their speed. This action influences the emergent behavior in real time. We need to be careful not to augment this distance over the sensing capabilities of each vehicle, fact that will preclude the formation of the platoon. On the other hand, too small a value could result in collisions. Other factors such as the number of lanes in the road also place constraints over this hyper-parameter. In this analysis we keep this distance between acceptable boundaries that do not compromise the behavior. Reducing the distance we induce a compact platoon that can overcome the obstacle easily. To induce proper trajectories, RSU_A establishes a destination point through R_4 to overcome the blockade smoothly. We are influencing the behaviors through its rules with the final objective of reducing the reaction time.

Figure 4.10 presents a temporal sequence on how the cars execute this primitive. The left figure presents the initial position of the platoon moving the highway in normal operation regime mode. The center figure shows the modified behavior after RSU_A has modified the separation hyper-parameter, the speed, and the destination target of the platoon. As you can observe, the platoon now is more compact and the vehicles move closer between them. The right figure shows the platoon overcoming the blockade previously notified to it. Finally, once the platoon totally surpassed the blockade RSU_B reestablishes the original separation distance, speed, and triggers the original target destination.

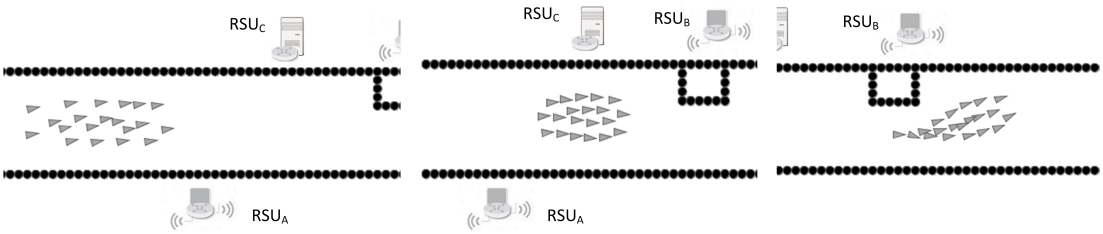

Figure 4.10: Sequential representation (from left to right) of a platoon executing an anticipated reaction to obstacles beyond the sensors range

The advantages of such a primitive includes a reduction of the time to overcome situations such as partial road blockades and to show how behaviors can be influenced by the contextual information. In this specific situation we achieve it through a hierarchy of RSUs although there are other solutions. The reaction time reduction comes from how vehicles face the blockade beyond their sensors range. If they are not prepared, a part of the platoon uses the blocked lanes in front of them. Once they sense that obstacle they will change their trajectory to avoid it, but they may have to wait till the vehicles on the clear lanes pass through it. The other possibility is even slower, when both vehicles intersect and the absolute velocity drastically diminishes. Instead, if the platoon is more compact and there are fewer cars on the blocked lanes the reaction time is smaller.

4.5.2.3 Concatenation of primitives to express complex behaviors

Judiciously chosen local rules, though simple to understand and implement, have the amazing capability of inducing behaviors not explicitly enunciated. Local rules are also flexible and expressive, enabling the creation of primitives through minor tweaking and additions, as shown in the previous Section. We observe that the exiting maneuver and obstacle anticipation primitives presented are built on top of a proto-primitive, the platoon formation. This observation suggests that complex behaviors can be achieved by chaining primitives.

The fairly extensive literature on self-driving vehicles that follow prescriptive designs [26] methodology can be leveraged in two ways: a) it suggests a list of primitives and complex behaviors to consider; b) the use cases can be used as baselines to compare with the HEB methodology.

4.6 Advances in HEB to Autonomous Vehicles

Autonomous Vehicles (AVs) is strongly positioning as a rapidly developing area, and it is achieving remarkable milestones day after day. Aerial (i.e. drones) and terrestrial vehicles (i.e. cars and trucks) are already perceiving their environment using a plethora of technologies (i.e. Lidar, cameras, infrared) to reach their destination safely while avoiding collisions [48].

Nowadays most approaches in the open literature focus on developing technologies from the single vehicle perspective. Corroborations can be found in the construction of high-definition maps [8] to navigate each car and the separation of onboard hardware and software platforms [7]. However, there is no unified theory or consensus on how to design and orchestrate such large systems with millions of vehicles and an uncountable number of external variables (i.e. pedestrians, driving rules, etc.).

HEB combines emergent behaviors with hierarchical decomposition to tackle this problem. HEB induces useful behaviors through local rules implemented at each AV rather than explicitly programming each action a vehicle must take in every circumstance [63]. Relying on emergent behaviors has major benefits. The first is the absence of highly complex algorithms. The second is HEB's intrinsic adaptivity to deal with unanticipated corner cases. These objectives are achieved by moving the decision-making capabilities to the vehicles and thus allowing them to take actions based on well understood rules.

The next logical step requires the development of a design methodology to build, evaluate, and run HEB-based solutions for AVS. Towards this goal, this section advances previous work on:

- Architectural foundations of the second level and its implications, with a focus on inter-level communication & locality and hierarchical relation between the rules, including the necessity of a leader and possible mechanisms to implement its selection.
- Demonstration of the robustness, flexibility, and smoothness of a HEB-based AV system.
- Case study to validate the previous points, incorporating new rules and experimental observations

4.6.1 First steps from high level concepts to a solid theory

Previous papers introduced the HEB concept [57], discussed the role played by Fog Computing [56], and explored through simulations a fairly rich set of emergent behaviors displayed under a variety of scenarios. The gained experience has

convinced us of the potential of HEB to become an important piece in advancing the introduction of autonomous vehicles at scale. A comprehensive theory of the phenomenology of collective behavior induced by local rules, and the interaction of the different elements within the system is required to consolidate HEB's ideas.

The ultimate goal is the development of a comprehensive theory that: a) captures the phenomenology of the collective behaviors induced by local rules; b) relates behaviors at different hierarchical levels; c) determines with high degree of confidence the range of validity of the approach. Such a theory would provide the foundational basis for the indispensable design methodology.

Toward this goal we focus in the following Sections on the the inter-level communication, the shaping of desired global behaviors through simple modifications of Reynolds's local rules, extensions of those rules to higher levels in the hierarchy, and key architectural attributes to quantify behaviors.

4.6.1.1 The vital role of communications

HEB relies on sensorial activity and communications to induce useful behaviors. A car not capable of knowing its environment (i.e. neighboring cars, obstacles) and its own condition (i.e. speed, position) will hardly produce either safe or interesting behaviors. Current technologies such as Lidar already support these needs, and its software integration is advancing quickly.

The behavior at the first hierarchical level is largely induced by local interactions between neighboring vehicles. Platoons emerge naturally from the application of first-level local rules. Second-level rules can induce behaviors that extend the scope of first-level rules (regulating, for instance the interactions between platoons). This requires mechanisms for both intra- and inter-level communication. Taking the platoon as the elemental unit, cars need to sense each other (intra) and simultaneously the platoon they form need to communicate with other platoons (inter), as reflected in Figure 4.11.

This broader locality translates into different detection ranges at the on-board "things". Despite this fact, the component that applies the different hierarchical rules and senses the environment remains unaltered, the vehicle. HEB elemental units can use a passive mechanism where each vehicle bases its behavior solely on

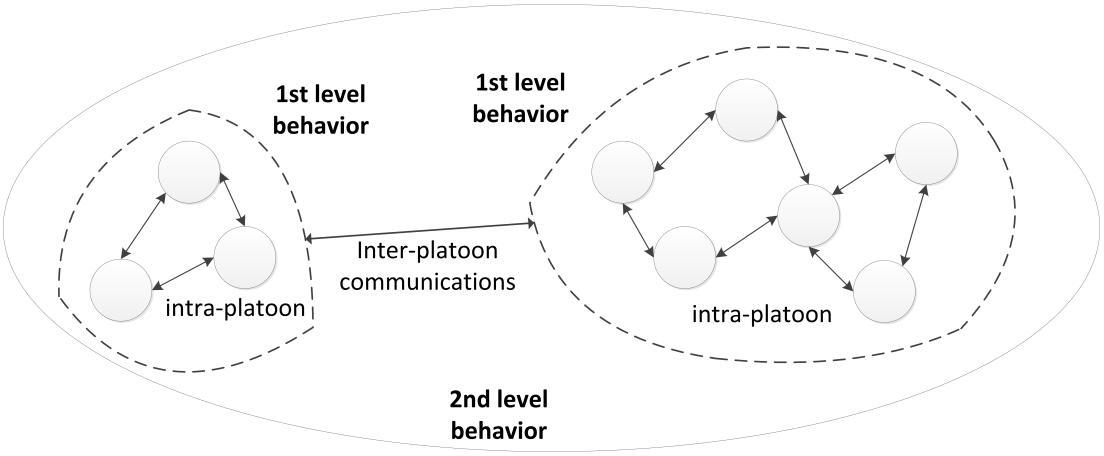

Figure 4.11: HEB's representation including intra- and inter-behavior sensing and two different level behaviors (1st level with dotted lines and 2nd level with continuous line)

the information sensed from its environment (also including other vehicles). Using this mechanism "things" become critical to achieve useful and secure behaviors. Another possibility uses active and direct mechanisms to code information between different entities and thus influencing the behavior directly. This direct strategy reduces the pressure placed upon "things", but moves the complexity to communication, synchronization, and coordination protocols.

A complementary technique relies on infrastructures such as Fog [15] to provide contextual information, enlarging the information scope of each vehicle beyond the on-board sensors. Hence, vehicle-to-infrastructure communications complements the aforementioned vehicle-to-vehicle capabilities. On one side this technique reduces the pressure placed upon "things" while on the other side moves away from the vehicles certain degree of independence to take their own decisions.

Emergent behavior organization Communication is essential to induce collective behaviors at every level of HEB's hierarchy. There are important differences between the first and the higher levels in the requirements and organization of the communication. At the first level this is strictly a local issue, that requires only V2V and V2X communication. In contrast, the second level requires also communication between platoons. There are several ways to approach the problem, and even variants within them. Let us outline the main ones.

The first solution calls for assigning a leader in each platoon. This leader, who could be either virtual [40] or physical [63], would drive the behavior of the platoon using the three Reynold's rules. In particular, the rest of the platoon would follow the trajectory taken by the leader. Platoons communicate through their leaders. It could be argued that depending on a leader goes somewhat against the grain of HEB's distributed principles. This suggests the consideration of distributed leadership schemes, which at the cost of higher level of local intelligence, could ensure scalability and autonomy in extreme scenarios.

4.6.1.2 Behavior inducement

Nature's goals (i.e., survival and reproduction) guide animal behavior. In the case of HEB we need to cast human and societal needs (for instance, reduction of driving time, and fuel consumption) into goals.

Once the goals are determined, the task is to find rules that induce behaviors to meet them. At this stage of development of HEB concepts, simulation appears as the right tool for the task. It enables exploration of the effect of some rules on others, and determination of how well the overall objectives are met. The quality of the results naturally depends on how well the simulator captures real life scenarios and their constraints.

Our previous sections explored the first level behavior of AVs as the result of applying Reynolds rules. The next step is to extend that exploration to second level behavior. The challenge is to find rules that induce desired behavior without destroying the first level platoon formation. Later we introduce rules that accomplish different AVs maneuvers.

We envision that as the theory underlying HEB consolidates, clever application of Machine Learning (ML) techniques will allow further extension of the rules, with an richer portfolio of emerging behaviors. For related example of how ML can automize the process, see Mataric [43, 44].

Single-level vs multi-level approach The multiplicity of hierarchical levels differentiates HEB with respect to preceding work in robotics focused on single level solutions [43]. The N-1th level enhances the scalability of the system and expands scope of achievable emergent behaviors.

We note that independently of the number of hierarchical levels, each vehicle is responsible for following a set of rules locally. Some rules are more critical than others (collision avoidance being a clear example). Hence, rules are organized in a hierarchy of dominance determined by their criticality. In practical terms, that dominance is expressed by the weights associated with each rule (called hyper-parameters). For instance, collision avoidance is a first level rule with weight larger than any other rule at any level, because safety is the dominant consideration.

At the single level the organization of the rules depend entirely on their relative weights. A multi-level design allows building richer behavior by combining first order ones. First level rules within a platoon are always active. In contrast, higher level rules are activated only when certain conditions are met, given the system the ability to incorporate "intelligent awareness".

For instance, a second level condition may trigger when two platoons become in close proximity, preventing V2V interactions between vehicles in different platoons. The vehicle checking the condition needs to sense one or more vehicles in its surroundings and two or more further away.

4.6.1.3 Behavior shaping

We study two approaches to the problem of shaping a given behavior induced by a set of rules: (i) slight modification of the rule/s, and (ii) tuning of the hyper-parameters. Through shaping behaviors a rule can go from being functionally correct but rough to smooth (e.g. making abrupt maneuvers more comfortable to the passengers), and from reaching a destination broadly defined to meeting a precisely defined one.

Rule/s modification Designing rules that express elaborate behaviors is an organic process. It starts with a core of elementary, local rules (Reynold's platoon formation) and compose them in complex chains that achieve specific objectives (compact the platoon, move aside, increase speed until a moving obstacle gets behind, etc.).

Let us consider some concrete instantiations of rules modification. We have already presented a destination rule [57] that directs vehicles to targeted points

without specifying the trajectory to follow. Each vehicle just knows its current position and its target destination. While this rule works fine at the vehicular level, the preservation of the platoon formation depends on how close the destination points specified for each individual car are, and the shape of the road. The rule can be retouched by applying it consecutively to a chain of segments, each one with its own target destination. The final coordinates remain unaltered, but the finer granularity ensures that vehicles remain close enough to induce platoon formation, hence maximizing traffic throughput [30].

A roadside infrastructure like Fog facilitates the implementation of this sequence of targets. Fog nodes' expanded scope allows the smart processing of target destinations and congestion information to dynamically building the chain of segments that compose the trajectories. An alternative implementation relies on V2V communication to exchange information regarding current positions, final destinations, and contextual information to determine intermediate target points. This latter alternative places stronger processing requirements on the vehicles on-board units (OBUs).

Hyper-parameters tuning The term hyper-parameter includes both the internal parameters of each rule (i.e. minimum separation distance) and the weights assigned to the rules. The core of hyper-parameters tuning takes place during the extensive experimentation phase in the controlled environment of a simulator. The simulator allows quantifying the behavior of experimental rules under a wide variety of scenarios. This leads to the acceptance of the rules and associated hyper-parameters, the tuning of the hyper-parameters, or, in extreme cases, downright rejection of the experimental rules. Note that the experimental rules examined not in isolation, but interacting with the whole set of rules.

4.6.1.4 Architectural attributes

Behaviors can be visually assessed, but their rigorous characterization requires the consideration of basic attributes that reflect the intent of the designer, and that can be translated into appropriate metrics.

- Sensitivity expresses the ability of the system to react to external stimuli. A HEB-based AV must be responsive to the environment, including obstacles,

and other vehicles. On the other hand, a hyper-sensitive vehicle may react too soon, or too violently. Consider a set of hyper-parameters that induce a desirable behavior. A system requiring infinite precision in the tuning of its parameters is impractical. In our case, the admissible loci of the hyper-parameters extend over a (not necessarily connected) hyper-volume.

- Expressiveness refers to HEB's ability to induce new desired behaviors extending the core framework through slight modifications of the existing rules or the addition of new ones, without affecting the existing ones. For instance, the three original Reynold's rules surpassed our expectations, because slight modifications enabled novel behaviors. The destination rule, which introduces a new behavior without affecting the formation of platoons, is a prime example.

- Smoothness relates to the user experience. Desired behaviors, including braking, acceleration, and change of direction shall not be brusque. For instance, a 180 degree turn on a highway is not smooth, and although the surrounding vehicles could respond adequately, such a maneuver will not contribute to the comfort of the passengers.

4.6.2 Multilevel interaction simulation and evaluation

4.6.2.1 Fundamentals

There are two types of terrestrial AVs represented by triangles of different colors (black and grey respectively) highlighting their direction. Obstacles define the shape of different scenarios such as highways or intersections, represented by black dots in the canvas.

Each vehicle implements the three original rules from Reynolds (R_1 Alignment, R_2 Separation, and R_3 Cohesion). In addition, there are two more rules conforming the basic set: R_4 and R_5. R_4 establishes a target destination point that each vehicle has to reach. R_5 performs the same operations as R_1, R_2, and R_3 but aggregated in a single rule and thus applies over platoons instead of single vehicles. The resulting rule is more complex than the three Reynolds rules but induces the same behavior, the creation of a platoon of platoons. Both rules

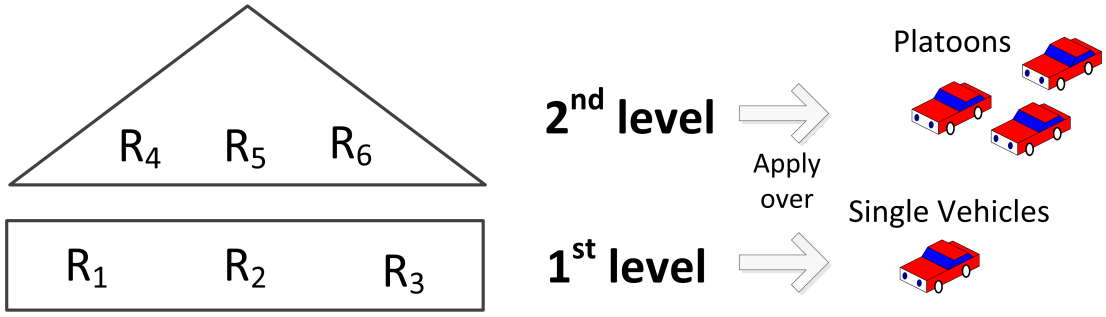

Figure 4.12: Depiction of a two level rule hierarchy and the entities at each level. R_1, R_2, and R_3 are the original Reynolds rules while R_5 is derivates from them. In contrast, R_4 and R_6 are not part of Reynolds work. Single vehicles constitute the entities of the first levels, while platoons constitute the entities of the second level (in the use case of this paper).

are implemented as second level rules but apply when different environmental conditions are satisfied. While R_4 acts upon individual platoons (defined by the presence of two or more vehicles within a certain distance), R_5 requires at least one platoon and one vehicle of another type to be applied.

On top of those, a new rule R_6 is added. R_6 is a complex 2nd level rule that guides interactions between different types of platoons circulating along a highway. It applies over vehicles when two types of platoons detect each other (what constitutes a 2nd level membership condition). Then R_6 acts upon both platoons. The approaching platoon changes their trajectory to focus on the left part of the road while the approached moves toward the right side. These turns are induced through a set of target destinations rather than being specifically programmed. The basic implementation of R_6 does not alter the velocity of each platoon. The rules' weights are set to prioritize the collision avoidance above the rest. Figure 4.12 represent the rules and the level they belong to.

4.6.2.2 2nd level: Platoon of platoons

Previous work explored the idea of platoons and how to enable them either by programming [72] or by inducement [57]. The same concept can be applied at a 2nd level resulting a platoon of platoons. To analyze this scenario we use R_5 that applies the original rules over platoons.

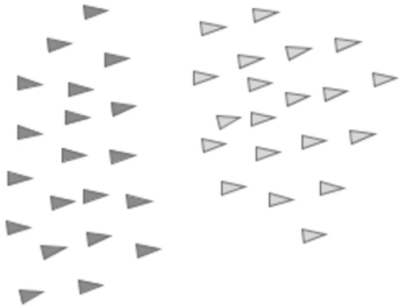

Figure 4.13: 2nd level behavior, platoon of platoons

Two 1st level platoons constitute a 2nd level platoon, that moves along the highway as depicted in Figure 4.13. Even though vehicles are responsible of determining its 2nd level membership to apply the proper rules through a condition (i.e. detect another platoon in its surroundings), the scalability is not compromised. The reason lies beneath the fact that not all vehicles are aware of the platoon behind them. Only those which sensor range allows them to detect those vehicles know about its existence. Even when the number of vehicles aware of the 2nd level is small, the imitation capabilities of the original rules makes the entire platoon to follow them, acting as a single 2nd level entity (similar effect to a wave propagation).

The separation distance between platoons constitutes a primary example of a behavior sensitivity analysis in conjunction with the sensors range (we suppose a fixed value for these ranges). When we analyze small increments of the separation distance, a 2nd level platoon still emerges. The limit in this case is fixed by the sensors capabilities. When the separation distance exceeds the sensors range the two platoons cannot "see" each other, preventing the emergent 2nd level platoon.

On the contrary, when the distance is reduced the same behavior is exhibited. The only major appreciation results when the separation distance is equal or smaller than the inter-platoon collision avoidance value. In this case the behavior is a larger and unique platoon. No collision occurs because R_2 prevails over the other rules. Facing these results, we can conclude that the sensitivity of this set

of rules is low and that the resultant behaviors are quite good, emerging for a wide range of hyper-parameter values.

Having a second level platoon results in an optimized traffic flow. Vehicles and infrastructure can adjust the separation distance and the velocity to adapt to road conditions without affecting 2nd level platoon entities.

4.6.2.3 Highway overcoming maneuver

This scenario highlights the overcoming maneuver optimization between platoons. For this purpose, we disable R_5 and activate R_6, the new rule designed to induce behaviors when two platoons face each other. Remember that by default R_6 does not modify the velocity.

1st level implementation The natural comparison to the aforementioned maneuver arises from a single level implementation. Each vehicle faces the situation alone rather than in conjunction with the rest of the platoon. This difference is reflected in the conditions required to apply R_6 as a 2nd level rule. It is only necessary to detect another type of vehicle ahead to start the maneuver but not to be a part of a platoon or detect another one. R_6 weight is set to the same value of the other rules except R_2.

Figure 4.14, Figure 4.15, and Figure 4.16 show a highway where a faster platoon (black) is about to reach a slower platoon (grey). When the rule is implemented as a first level rule, the behavior obtained is not smooth as an architect would like. However, the resultant behavior is modified and the interaction suffer a small optimization. The slower vehicles move towards the right side (as individual elements, any action is taken as a platoon). This fact is exploited by the faster vehicles that move to the left side for an overtaking maneuver.

As expected, in most of the simulations single vehicles cross the slower platoon instead of continuing attached to their original behavior. This action could put in danger the safety of both platoons, plus implies a strong modification of the first level behaviors (rupture of the platoon behavior). The faster platoon faints to overtake the other.

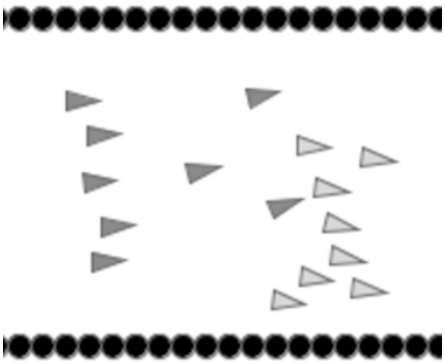

Figure 4.14: First temporal instance of a 1st level based overcoming maneuver behavior

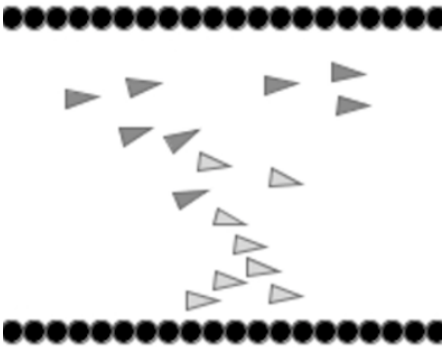

Figure 4.15: Second temporal instance of a 1st level based overcoming maneuver behavior

2nd level implementation Trying to avoid those situations, R_6 is implemented as a second level rule. As mentioned earlier, the key differentiation between a 1st and a 2nd level implementation lies beneath the applying conditions for the rule. In R_6, these conditions are to be part of a platoon and to detect the presence of a different platoon (or part of it) ahead or behind. In consequence, each vehicle acquires membership awareness to apply 2nd rules as part of a platoon. Despite this fact, vehicles may not know the total size of the platoon to

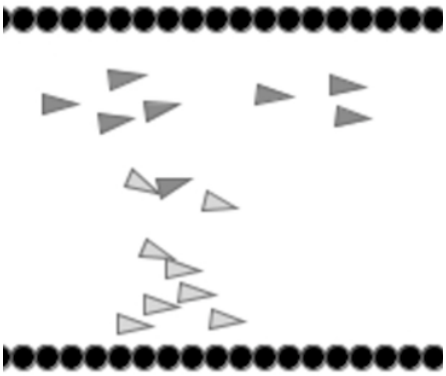

Figure 4.16: Third temporal instance of a 1st level based overcoming maneuver behavior

not compromise its scalability. R_6 now influences the vehicles' trajectories only when those conditions are satisfied.

Although it may seem a simple modification, the fact that vehicles try to stay within its platoon rather than taking individual actions results in the absence of crossing cars between platoons. Figure 4.17, Figure 4.18, and Figure 4.19 depict a temporal sequence of images to capture the overtaking maneuver.

Different observations arise from these figures. First, the shape of the platoon changes, but this is not a problem since it was not specifically programmed. In fact, this behavior is desirable because thinner and longer shapes result, facilitating the maneuvers. Second, the destination target modification from a single point to a set of points applies only when R_6 conditions are satisfied (be part of a platoon and detect another one). Within the context of the highway analysis, this set of targets is determined by a vertical offset to induce that movement to the sides (left or right) over a spatial sequence in the horizontal dimension. If only the vertical offset is applied together with a single destination point the behavior reaction was not fast enough to ensure a smooth maneuver.

Although the objective was accomplished with this rule, an architect can explore how to enhance the resultant behavior through the modification of rule. This process is referred to as "Behavior Shaping", as explained in Section 4.6.1.3.

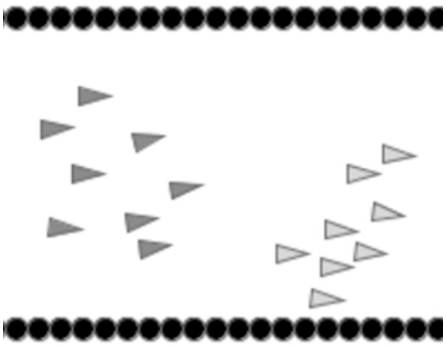

Figure 4.17: First temporal instance of a 2nd level based overcoming maneuver behavior

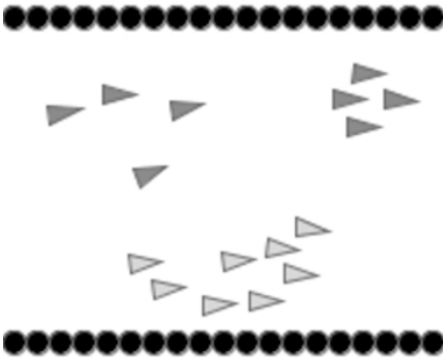

Figure 4.18: Second temporal instance of a 2nd level based overcoming maneuver behavior

4.6.2.4 Behavior shaping: a practical example

Since formations revealed as a way to optimize interactions, this the first technique analyzed. Using a rectangular formation induced through the initial positions of the AVs resulted in smoother maneuvers in both the intersection and the highway scenarios (we could insert a figure).

An important remark is how adaptive the rules are to enable more behaviors,

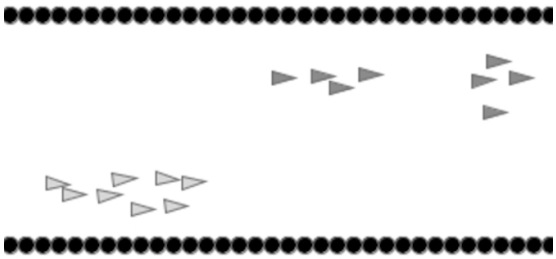

Figure 4.19: Third temporal instance of a 2nd level based overcoming maneuver behavior

in this case through a formation. Instead of designing a new rule or modifying the existing ones, setting studied initial positions to each vehicle at the beginning of the simulation created the formation. Later, applying the three original rules resulted in the maintenance of the formation for their entire trajectory even when facing other platoons.

Another technique alters the hyper-parameters of the platoons. For instance, if the intra-platoon separation distance is modified upon the detection of the other platoon the total surface occupied by each platoon is smaller and in consequence the space for the overtaking maneuver becomes larger, reducing the risk of potential collisions or strange maneuvers (i.e. 90 degrees turns). Similarly to the target destination modification, hyper-parameters are modified upon the fulfillment of the 2nd level membership conditions. When the faster platoon has completed the overtaking, both platoons can return to their original separation distances because 2nd level conditions are no longer satisfied.

The platoon velocity is a different parameter to consider. In this case, upon the detection of a slower platoon ahead the faster platoon can moderate its speed to have more control over its maneuvers. On the contrary, the slower platoon can momentarily accelerate to move aside faster, leaving more space for the overtaking maneuver. One possibility is to use Fog nodes to provide contextual information to the AVs, contributing to determine the N-level membership conditions. These external nodes could also act directly upon the vehicles exploiting its larger visibility.

Till now vehicles had a simple second level condition (i.e. be part of platoon and detect another one), but more detailed conditions can be studied based on their impact over the resultant behaviors. For instance, within a condition we can have different phases based on contextual information, inducing more robust behaviors.

For example, we can use a three phase illustrative example. The first phase is activated upon the detection of the slower platoon ahead. In that moment this phase triggers the aside movements. The second phase is activated when the distance between platoons is smaller. In this situation, the slower platoon accelerates for a moment and the faster platoon maintain its speed. This momentary acceleration does not compromise the overtaking maneuver thanks to the target modification and its short burst nature. The third and final phase is activated when the distance between platoons is negligible and both have a clear path ahead, accelerating the faster platoon to complete a faster overtaking maneuver. At the end, when this maneuver is completed and the distance between platoons is larger, the 2nd level conditions are no longer satisfied and default conditions are restored. In this example only the inter-platoon distance has been considered as contextual information, but many more parameters can be used.

The aforementioned techniques revealed how small modifications with low complexity have a great effect on induced behaviors. In future implementations, machine learning techniques can guide the behavior shaping process in its different approaches (hyper-parameters and phases), as proven by the first level approach.

4.7 Summary

Many IoT services and applications are intrinsically large and complex, fully qualifying as ULSS. In this chapter we address the problem of architecting and orchestrating ULSS. We propose a Hierarchical Emergent Behaviors (HEB) approach that borrows concepts from the fields of emergent behaviors and hierarchical decomposition. Furthermore, this chapter has made some solid steps along the goal of systematizing HEB's concept into a solid design methodology. In particular, we have advanced the understanding of the communication mechanisms required

between the different hierarchical levels, discussed the desirable attributes of vehicular behaviors, and demonstrated through simulations how simple second level rules can enrich the space of emergent behaviors.

By fusing emergent behaviors and hierarchical organization concepts, our architecture (HEB) uses only a minimal sets of engagement rules to achieve the desired behaviors without the need to explicitly program for every scenario. These techniques provide the system with less developer complexity and a natural ability to scale and adapt in dynamic environments. In order to implement this approach, we added a new layer to the traditional IoT architectural stack that incorporates the local rules of engagement. The autonomous vehicles case study was provided to illustrate the main properties and behaviors of "things" using the HEB approach.

Chapter 5

Conclusions and Future Work

This thesis analyzes the orchestration, management, and scalability issues of IoT ULSS and proposes a new technique to solve them.

We introduced the key concept of Hierarchical Emergent Behaviors (HEB) that combines the advantages of emergent behaviors and hierarchical decomposition. HEB imposes lightweight rules at the "things" to induce useful behaviors rather than explicitly program them. These rules exploit the contextual information available at the device level, reason why these devices can take decisions locally. We showed how HEB can improve the scalability of an autonomous vehicle system while reducing the design and the development complexity. We believe that HEB is the way to enable and build IoT ULSS with millions of devices such as autonomous vehicles.

5.1 Broader Impact

IoT systems are expected to improve human life in many aspects, from healthcare to transportation. It is of outrageous importance that we identify the critical areas that preclude its explosion and provide with efficient solutions that empower not only those systems but that also enable their interoperability.

This intersystem interaction is the base to create IoT ULSS with richer functionalities than the ones each system can provide in isolation. However, this area is still in its infancy. And despite the great public interest in having IoT systems such as smart homes or autonomous vehicles numerous problems have been arisen

and yet there is no consensus in areas such as privacy and security, even much less regarding the scalability and the management of ULSS. To the best of our knowledge, the open literature focus on single elements of these systems rather than seeing them as a massive group of devices (i.e. car).

The content of this thesis attacks the area of IoT ULSS to make them scalable and yet adaptive systems in a way to contribute to IoT's expansion and wider deployability.

5.2 Future Work

Some of the contributions of this thesis may be extended. In this section we provide some guidelines.

iQ: a simulation methodology based on queue models and statistical information. The first step focuses on emulating more components of the core model, such as floating point capabilities and the TLBs modules. Future work also includes the implementation of a technique to simulate the dependency chains that, for now, provoke errors in the gcc benchmark. Another enhancement is the implementation of power models triggered by the time the different queues and servers are really working. Then, architects will have available functions to determine the cost of their possible processor's implementations.

Further steps are the design and definition of multicore processor models, including the network on chip which translates nicely to our queue models and statistically driven events. Lastly, studies where both the processor and the network are simulated in a single environment will be performed to optimize the performance and cost of the global solution (currently, the problem is architectural simulators have poor network modeling and network simulators normally use traffic generators to feed the network).

Enhancements to Fog's architecture to enable the generic IoT platform. We have outlined a program to enhance Fog's architecture complementing the Cloud, opening three main areas for future research. Each area addresses different pillars of the generic Fog Architecture that entities such as the Open Fog Consortium are consolidating. The first line of research focuses on orchestration policies to decide which application run on the Fog layers (based on their

requirements, current conditions, etc.), and how these policies are enforced on Fog's heterogeneous nodes.

The second area focuses on constellations and their virtualization foundations. These groupings are influenced by networks conditions the infrastructure capabilities. This fact suggests simulation as a way to evaluate a rich set of scenarios where constellation design is enhanced. The last suggested area involves FFVs and the environment required to enable them. First, we need to develop the framework supporting the functions and acting as an interface between the different entities (ISPs, application developers, etc.). Later the actors need to deploy and offer "things" functionalities, creating a pool of resources from where the FFVs can emerge. Finally, the application developers can use those functions and enable new services and applications.

HEB: a design methodology based on emergent behaviours and hierarchical decomposition for ULSS. We have outlined a program to tackle the ULSS IoT challenge, opening four key areas for future research. The first line of research focuses on extensive realistic simulations of well-designed scenarios as a design tool. The emulation is the experimental design platform that allows architects to determine which rules are more likely to produce interesting behaviors, tune their hyper-parameters, and assess the performance in dynamic environments.

Machine Learning (ML) of rules that generate emergent behaviors is the second line of research. We note the strong link between the emergent behavior approach and ML, in both circumventing the need of explicit programming. ML can become valuable in choosing and tuning the hyper- parameters of local rules, and even identifying new useful rules. The experimental platform randomly sweeps the scenario space, and for each scenario ML tunes the hyper-parameters, or otherwise introduces new rules.

The last suggested research line involves security and reliability. Emergent behaviors bring new strengths: the system that can adapt itself as a whole in the face of new circumstances, including failures. On the flip side, there are new vulnerabilities to overcome, such as those that can be introduced through rogue "things", or by altering the hyper-parameters of the local rules.

5.3 Further Acknowledgements

This thesis was supported by a Doctoral Scholarship provided by Fundación La Caixa, to whom the author of the thesis is very grateful for their incredible support over this journey. This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493) and by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P).

Chapter 6

Publications

The contents of this thesis led to the following publications:

D. Roca, D. Nemirovsky, M. Nemirovsky, R. Milito and M. Valero, **"Emergent Behaviors in the Internet of Things: The Ultimate Ultra-Large-Scale System,"** in IEEE Micro, vol. 36, no. 6, pp. 36-44, Nov.-Dec. 2016.

D. Roca, J. V. Quiroga, M. Valero and M. Nemirovsky, **"Fog Function Virtualization: A flexible solution for IoT applications,"** IEEE Second International Conference on Fog and Mobile Edge Computing (FMEC), Valencia, Spain, 2017, pp. 74-80.

D. Roca, R. Milito, M. Nemirovsky, and M. Valero, **"Tackling IoT Ultra Large Scale Systems: Fog Computing in Support of Hierarchical Emergent Behaviors"**. Fog Computing in the Internet of Things (pp. 33-48). Springer International Publishing, 2017

D. Roca, D. Nemirovsky, M. Nemirovsky, M. Casas, M. Moreto, and M. Valero, **"iQ: An Efficient and Flexible Queue-based Simulation Framework"**, in IEEE 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (Mascots), Sept. 2017

6. PUBLICATIONS

These other papers were published during graduate studies but are out of the scope of this thesis:

S. Abadal, B. Sheinman, O. Katz, O. Markish, D. Elad, Y. Fournier, D. Roca, M. Hanzich, G. Houzeaux, M. Nemirovsky, E. Alarcon, and A. Cabellos-Aparicio, "Broadcast-Enabled Massive Multicore Architectures: A Wireless RF Approach," in IEEE Micro, vol. 35, no. 5, pp. 52-61, Sept.-Oct. 2015.

K. Katrinis, G. Zervas, D. Pnevmatikatos, D. Syrivelis, T. Alexoudi, D. Theodoropoulos, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "On interconnecting and orchestrating components in disaggregated data-centers: The dReDBox project vision," 2016 European Conference on Networks and Communications (EuCNC), Athens, 2016, pp. 235-239.

K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends, "Rack-scale disaggregated cloud data centers: The dReDBox project vision," 2016 Design, Automation and Test in Europe Conference and Exhibition (DATE), Dresden, 2016, pp. 690-695.

H. Meyer, J. C. Sancho, J. V. Quiroga, F. Zyulkyarov, D. Roca and M. Nemirovsky, "Disaggregated Computing. An Evaluation of Current Trends for Datacentres." Procedia Computer Science 108 (2017): 685-694.

Appendix A

Flock and Platoon background

A.1 Rules

HEB relies on local engagement rules to induce behaviors at the vehicle level. These rules determine the behavior of each car when facing different situations such as encounters between them or when facing an obstacle. We took the original flocking implementation code contained in the Processing environment and extended it to add or modify the rules and how they are applied. Saber described the mathematical formulation for the rules [60]. In the next Sections we explain these rules and their effect on the behavior.

A.1.1 Background

Each vehicle applies the implemented set of rules. These rules are local in the sense that a vehicle is not aware of the size of the platoon. Vehicles apply the rules over their neighbors. The locality of the rules is key to ensure the scalability of the behavior and keep programming efforts low. For instance, in a flock with thousands of cars, each vehicle applies the rules only to its neighbors. These neighbors are sensed through the on-board sensors, and normally due to physical constraints its number is limited to 8 or 9.

Then, each vehicle has a bubble around it and only those vehicles within this range are used to compute the rules. In other words, these vehicles determine the engagements that drive the induced behaviors. Since each vehicle applies the

same principle, and effect on one side of a platoon is transmitted over to the other side by the mimic effect of certain rules (as a wave).

An important remark is that not all the vehicles need to have the same set of rules. While certain rules are necessary to ensure certain level of safety (i.e. collision avoidance) others are "optional". In consequence, it is vital that possible interactions between these sets of rules are simulated to study and analyze the behaviors that result from those interactions before they are implemented in a real scenario.

A.1.2 R_1, Alignment

R_1, nicknamed alignment, makes the vehicle to match the average heading of the neighbors. This rule results in all neighboring vehicles moving with the same direction. Figure A.1 illustrates this situation.

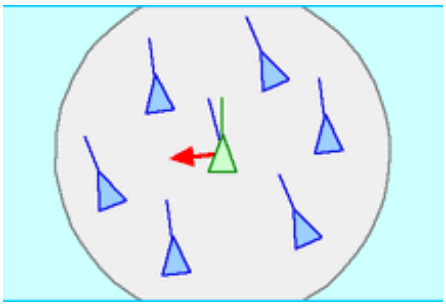

Figure A.1: R_1, Aligment [1]

A.1.3 R_2, Separation

R_2, nicknamed separation, provokes that the vehicle keeps a certain distance with its neighboring vehicles and other obstacles. This rule results in a collision avoidance mechanism, critical to ensure basic safety trajectories. The main parameter within this rule is the desired separation distance between elements of the system. Figure A.2 illustrates this situation.

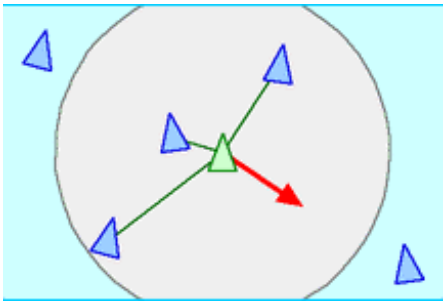

Figure A.2: R_2, Separation [1]

A.1.4 R_3, Cohesion

R_3, nicknamed cohesion, provokes that the vehicle stays close to its neighbors.This rule results in the vehicles moving together. Figure A.3 illustrates this situation. R_2 and R_3 apply an attraction/repulsion force where the cars want to stay close to each other but not too close. They are responsible of the grouping between vehicles.

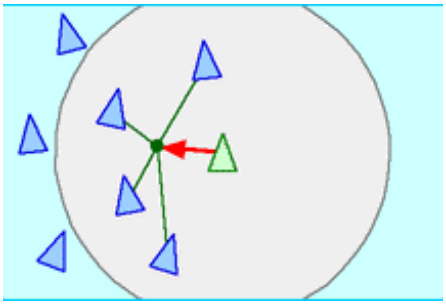

Figure A.3: R_3, Cohesion [1]

A.1.5 R_4, Destination

R_4, nicknamed destination, makes that the vehicle moves towards a target destination. This rule does not specify a complete trajectory rather than an origin and a destination, leaving decisions to the vehicles themselves based on the local information they have and what they sense.

A.1.6 R_5, 2nd level platoon

R_5 combines the three original Reynolds' rules into a single one. This fact highlights that different rules can have different levels of complexity. The result of this rule is the inducement of a 2nd level platoon when the conditions allow for it, a platoon of platoons. R_5 enforces that platoons do not collide, stay close to each other and move at the same speed similarly to how R_1, R_2, and R_3 enforce it between vehicles.

A.1.7 R_6, Overcoming maneuver

R_6 defines the interaction between two platoons moving at different speeds alongside a highway. Depending on the platoon positioning (pursuer and pursued) there is a set of actions defined. For instance, the pursuer moves to the left side of the road to make the maneuver smoother while the pursued moves towards the right side of the road.

A.2 Application of the rules

Vehicles are applying their set of rules constantly to determine what their trajectory is. It is an infinite loop that keeps updating their trajectory based on what they sense.

It is important to explain where the consciousness of the level resides. It is the vehicle that knows in what level is and executes the appropriate rules. For instance, a vehicle that moves inside a platoon of platoon is executing the 2nd level rule (R_5), but also the first level rules (R_1, R_2, and $R_{5=3}$). A second level behavior cannot destroy a first level behavior. This way rules in different hierarchical levels build on top of each other rather than all being at the same level.

A.2.1 Weights

Different weights ponder the result of the rules to determine both their criticality and impact on the induced behavior. In this case, to ensure the safety of the

resultant behavior R_2 has the larger weight to always ensure that no collision happens.

These weights can also be used to define a hierarchy among the rules (between a single HEB level or multiple levels).

A.2.2 Sum of vectors

Once all the rules are computed, vehicles have different vectors with updates to their trajectory. Then, each vehicle computes the sum of the vectors to determine which is the trajectory change to be performed. Note that different rules can give opposite vectors. In this case, the resultant vector may not modify the trajectory of the vehicle. If this happens, weights can help to ensure reliable movements, such as the vector component to avoid an obstacle is larger than that of following neighboring vehicles. This situation results in the vehicle moving to avoid the obstacle.

References

[1] Craig reynolds website, [link]. xi, 106, 107

[2] Heb code repository, [link]. 8

[3] iq code repository, [link]. 8

[4] Its vehicle to infrastructure resources, [link]. 75

[5] Openfog consortium reference architecture, [link]. 40

[6] Processing simulation framework, [link]. 67

[7] Software defined vehicles, [link]. 83

[8] Waymo maps for self-driving cars, [link]. 83

[9] Sarfraz Alam, Mohammad MR Chowdhury, and Josef Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *IEEE International Conference on NESEA*, 2010. 43

[10] Arwa Alrawais, Abdulrahman Alhothaily, Chunqiang Hu, and Xiuzhen Cheng. Fog computing for the internet of things: Security and privacy issues. *IEEE Internet Computing*, 21(2):34–42, 2017. 43

[11] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 2010. 11, 55

[12] Sion Berkowits. Pin-a dynamic binary instrumentation tool, 2012. 21

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011. 9, 15

[14] Eric Bonabeau, Guy Theraulaz, Jean-Louls Deneubourg, Serge Aron, and Scott Camazine. Self-organization in social insects. *Trends in Ecology & Evolution*, 1997. 58

[15] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the 1st edition of the workshop on Mobile cloud computing*. ACM, 2012. 2, 10, 39, 70, 85

[16] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 SC*, page 52. ACM, 2011. 15, 29

[17] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, C Cui, H Deng, et al. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, 2012. 48

[18] Jeanine Cook, Jonathan Cook, and Waleed Alkohlani. A statistical performance model of the opteron processor. *ACM SIGMETRICS Performance Evaluation Review*, 2011. 37

[19] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. (March), 2014. 21

[20] Scott de Deugd, Randy Carroll, Kevin Kelly, Bill Millett, and Jeffrey Ricker. Soda: Service oriented device architecture. *IEEE Pervasive Computing*, 2006. 48

[21] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. *Proceedings of PACT*, 2001. 15

[22] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 2011. 39

[23] Hassan Farhangi. The path of the smart grid. *Power and energy magazine*, 2010. 51

[24] Agner Fog. The microarchitecture of intel, amd and via cpus/an optimization guide for assembly programmers and compiler makers, 2012. 21, 23, 24

[25] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *2008 Grid Computing Environments Workshop*. IEEE, 2008. 45

[26] Emilio Frazzoli, Munther A Dahleh, and Eric Feron. Real-time motion planning for agile autonomous vehicles. *Journal of Guidance, Control, and Dynamics*, 25(1):116–129, 2002. 82

[27] Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *2014 IEEE World Forum on Internet of Things,*. 66

[28] Thomas Grass, Alejandro Rico, Marc Casas, Miquel Moreto, and Eduard Ayguadé. Taskpoint: Sampled simulation of task-based programs. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 296–306. IEEE, 2016. 37

[29] Anthony Gutierrez, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. Sources of error in full-system simulation. *Proceedings of ISPASS*, 2014. 26, 29

[30] Randolph Hall and Chinan Chin. Vehicle sorting for platoon formation: Impacts on highway entry and throughput. *Transportation Research Part C: Emerging Technologies*, 13(5):405–420, 2005. 78, 88

[31] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006. 22

[32] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the 2nd ACM workshop on Mobile Cloud Computing*, 2013. 44

[33] Kenneth Hoste and Lieven Eeckhout. Microarchitecture independent workload characterization. In *IEEE Micro*, 2007. 22

[34] Ann Hsu, Farokh Eskafi, Sonia Sachs, and Pravin Varaiya. Design of platoon maneuver protocols for ivhs. *California Partners for Advanced Transit and Highways (PATH)*, 1991. 75

[35] Raj Jain and Sudipta Paul. Network virtualization and software defined networking for cloud computing: a survey. *Communications Magazine, IEEE*, 2013. 43, 44

[36] Henrik Jeldtoft Jensen. *Self-organized criticality: emergent complex behavior in physical and biological systems*. Cambridge university press, 1998. 56

[37] Alex Kushleyev, Daniel Mellinger, Caitlin Powers, and Vijay Kumar. Towards a swarm of agile micro quadrotors. *Autonomous Robots*, 2013. 56

[38] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. *Workload characterization of emerging computer applications*, 2001. 9, 15

[39] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*. ACM, 2006. 37

[40] Naomi Ehrich Leonard and Edward Fiorelli. Virtual leaders, artificial potentials and coordinated control of groups. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 3, pages 2968–2973. IEEE, 2001. 86

[41] Mark W Maier. Architecting principles for systems-of-systems. In *INCOSE International Symposium*. Wiley Online Library, 1996. 56

[42] Maja J Mataric. Designing emergent behaviors: From local interactions to collective intelligence. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, 1993. 58, 64

[43] Maja J Mataric. Interaction and intelligent behavior. Technical report, DTIC Document, 1994. 86

[44] Maja J Matarić. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83, 1997. 86

[45] David Meisner and TF Wenisch. Stochastic queuing simulation for data center workloads. *Exascale Evaluation and Research Techniques Workshop*, 2010. 16, 37

[46] David Meisner, Junjie Wu, and Thomas F. Wenisch. Bighouse: A simulation infrastructure for data center systems. *ISPASS*, 2012. 37

[47] José Melo, Andrew Naftel, Alexandre Bernardino, and José Santos-Victor. Detection and classification of highway lanes using vehicle motion trajectories. *IEEE Transactions on intelligent transportation systems*, 7(2):188–200, 2006. 74

[48] Amir Mukhtar, Likun Xia, and Tong Boon Tang. Vehicle detection techniques for collision avoidance systems: A review. *IEEE Transactions on Intelligent Transportation Systems*, 16(5):2318–2338, 2015. 82

[49] Takayuki Nishio, Ryoichi Shinkuma, Tatsuro Takahashi, and Narayan B Mandayam. Service-oriented heterogeneous resource sharing for optimizing service latency in mobile cloud. In *Proceedings of the first international workshop on Mobile cloud computing & networking*. ACM, 2013. 47

[50] Linda Northrop, Peter Feiler, Richard P Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, et al. Ultra-large-scale systems: The software challenge of the future. Technical report, DTIC Document, 2006. 12, 55

[51] S. Nussbaum and J.E. Smith. Modeling superscalar processors via statistical simulation. *Proceedings of PACT*, 2001. 15, 37

[52] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: Micro architectural systems simulator. In *ISCA tutorial 6*, 2012. 9, 29

[53] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by internet of things. *Transactions on Emerging Telecommunications Technologies*, 25(1):81–93, 2014. 43

[54] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH computer graphics*. ACM, 1987. 58, 71

[55] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramirez, and Mateo Valero. Trace-driven simulation of multithreaded applications. In *Proceedings of ISPASS*, pages 87–96. IEEE, 2011. 9, 15

[56] Damian Roca, Rodolfo Milito, Mario Nemirovsky, and Mateo Valero. Tackling iot ultra large scale systems: Fog computing in support of hierarchical emergent behaviors. In *Fog Computing in the Internet of Things*, pages 33–48. Springer International Publishing, 2017. 14, 83

[57] Damian Roca, Daniel Nemirovsky, Mario Nemirovsky, Rodolfo Milito, and Mateo Valero. Emergent behaviors in the internet of things: The ultimate ultra-large-scale system. *IEEE Micro*, 36(6):36–44, 2016. 14, 83, 87, 90

[58] Damian Roca, Josue V Quiroga, Mateo Valero, and Mario Nemirovsky. Fog function virtualization: A flexible solution for iot applications. In *Fog and Mobile Edge Computing (FMEC), 2017 Second International Conference on*, pages 74–80. IEEE, 2017. 14

[59] Damian Roca, Josue V Quiroga, Mateo Valero, and Mario Nemirovsky. iq: an efficient and flexible queue-based simulation framework. In *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2017. 14

[60] Reza O Saber. A unified analytical look at reynolds flocking rules. Technical report, CALIFORNIA INST OF TECH PASADENA CONTROL AND DYNAMICAL SYSTEMS, 2003. 105

[61] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *Proceedings of ISCA*, 2013. 15, 29

[62] Kengo Sasaki, Naoya Suzuki, Satoshi Makido, and Akihiro Nakao. Vehicle control system coordinated between cloud and mobile edge computing. In *Society of Instrument and Control Engineers of Japan (SICE), 2016 55th Annual Conference of the*, pages 1122–1127. IEEE, 2016. 70

[63] Martin Saska, Vojtěch Vonásek, Tomáš Krajník, and Libor Přeučil. Coordination and navigation of heterogeneous mav–ugv formations localized by a hawk-eye-like approach under a model predictive control scheme. *The International Journal of Robotics Research*, 2014. 61, 83, 86

[64] Sangjin Shin, Seungmin Seo, Sungkwang Eom, Jooik Jung, and Kyong-Ho Lee. A pub/sub-based fog computing architecture for internet-of-vehicles. In *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, pages 90–93. IEEE, 2016. 70

[65] Herbert A Simon. *The architecture of complexity*. Springer, 1991. 59

[66] Ivan Stojmenovic and Sheng Wen. The fog computing paradigm: Scenarios and security issues. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pages 1–8. IEEE, 2014. 40

[67] Jeffrey Taft and P De Martini. Ultra large-scale power system control architecture. *Cisco Systems*, 2012. 60

[68] T Tsuei and Wayne Yamamoto. A processor queuing simulation model for multiprocessor system performance analysis. In *Proc. of 5th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 58–64, 2002. 37

[69] Thin Fong Tsuei and Wayne Yamamoto. Queuing simulation model for multiprocessor systems. *Computer*, 2003. 16, 37

[70] Sam Van den Steen, Sander De Pestel, Moncef Mechri, Stijn Eyerman, Trevor Carlson, David Black-Schaffer, Erik Hagersten, and Lieven Eeckhout. Micro-architecture independent analytical processor performance and power modeling. In *Proceedings of ISPASS*. IEEE, 2015. 29, 30

[71] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014. 47

[72] Pravin Varaiya. Smart cars on smart roads: problems of control. *Automatic Control, IEEE Transactions on*, 1993. 58, 71, 73, 90

[73] András Varga. The OMNeT++ discrete event simulation system. *ESM)*, 2001. 21

[74] Zhenyu Wen, Renyu Yang, Peter Garraghan, Tao Lin, Jie Xu, and Michael Rovatsos. Fog orchestration for internet of things services. *IEEE Internet Computing*, 21(2):16–24, 2017. 46

[75] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH computer architecture news*, 1995. 29

[76] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of ISCA.*, 2003. 15

[77] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003. 9, 37

[78] Marcelo Yannuzzi, R Milito, René Serral-Gracià, D Montero, and Mario Nemirovsky. Key ingredients in an iot recipe: Fog computing, cloud computing, and more fog computing. In *IEEE 19th International Workshop on CAMAD*, 2014. 39

[79] Marcelo Yannuzzi, Frank van Lingen, Anuj Jain, Oriol Lluch Parellada, Manel Mendoza Flores, David Carrera, Juan Luis Pérez, Diego Montero, Pablo Chacin, Angelo Corsaro, et al. A new era for cities with fog computing. *IEEE Internet Computing*, 21(2):54–67, 2017. 40, 51

[80] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015. 47

[81] Ruken Zilan, Javier Verdú, and J García. An abstraction methodology for the evaluation of multi-core multi-threaded architectures. *IEEE 19th MASCOTS*, 2011. 16, 37